AD/A-005 413

FORMAL SEMATICS OF LISP WITH APPLICATIONS TO PROGRAM CORRECTNESS

Malcolm C. Newey

Stanford University

Prepared for:

Advanced Research Projects Agency

January 1975

DISTRIBUTED BY:

NTIS
National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		**READ INSTRUCTIONS BEFORE COMPLETING FORM**
1. REPORT NUMBER STAN-CS-75-475	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD/A-005413
4. TITLE (and Subtitle) FORMAL SEMANTICS OF LISP WITH APPLICATIONS TO PROGRAM CORRECTNESS.		5. TYPE OF REPORT & PERIOD COVERED technical, Jan. 1975
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-75-475
7. AUTHOR(s) Malcolm C. Newey		8. CONTRACT OR GRANT NUMBER(s) DAHC 15-73-C-0435
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Computer Science Department Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/IPT Attn: S. D. Crocker 1400 Wilson Blvd., Arlington, Va. 22209		12. REPORT DATE Jan. 1975
		13. NUMBER OF PAGES 185
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR Representative : Philip Surra Durand Aeronautics Bldg., Rm. 165 Stanford University Stanford, California 94305		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Releasable without limitations on dissemination.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Described are some experiments in the formalisation of the LISP programming language using LCF (Logic for Computable Functions.). The bulk of each experiment was concerned with applying the formalisation to proofs of correctness of some interesting LISP functions using Milner's mechanised version of LCF.

A definition of Pure LISP is given in an environment which includes an axiomatisation of LISP S-expressions. A primitive theory (a body of theorems in LCF) of Pure LISP is derived and is applied to (continued) proving the correctness of some simple LISP functions using the LCF proof checking system. A proof of correctness of McCarthy's interpreter is described and a machine checked proof of the partial correctness is outlined.

A more substantial subset of LISP and a subset of LAP (a LISP-oriented assembly language for the PDP-10 computer) were formalised and simple theories for the two languages were developed with computer assistance. This was done with a view to proving the correctness of a compiler, written the LISP subset, which translates LISP functions to LAP subroutines. The coarse structure of such a compiler correctness proof is displayed.

063154

Stanford Artificial Intelligence Laboratory
Memo AIM-257

Computer Science Department
Report No. STAN-CS-75-475

JANUARY 1975

AD A005413

Formal Semantics of LISP With Applications to Program Correctness

by

Malcolm C. Newey

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

DDC
RECEIVED
FEB 19 1975
E

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

188

Stanford Artificial Intelligence Laboratory
Memo AIM-257

JANUARY 1975

Computer Science Department
Report No. STAN-CS-75-475

Formal Semantics of LISP With Applications to Program Correctness

by

Malcolm C. Newey

D D C
FEB 19 1975
E

ABSTRACT

Described are some experiments in the formalisation of the LISP programming language using LCF (Logic for Computable Functions). The bulk of each experiment was concerned with applying the formalisation to proofs of correctnes of some interesting LISP functions using Milner's mechanised version of LCF.

A definition of Pure LISP is given in an environment which includes an axiomatisation of LISP S-expressions. A primitive theory (a body of theorems in LCF) of Pure LISP is derived and is applied to proving the correctness of some simple LISP functions using the LCF proof checking system. A proof of correctness of McCarthy's interpreter is described and a machine checked proof of the partial correctness is outlined.

A more substantial subset of LISP and a subset of LAP (a LISP-oriented assembly language for the PDP-10 computer) were formalised and simple theories for the two languages were developed with computer assistance. This was done with a view to proving the correctness of a compiler, written the LISP subset which translates LISP functions to LAP subroutines. The coarse structure of such a compiler correctness proof is displayed.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435 . The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22151.

Reproduced from
best available copy.

Particular attention is paid, in describing the experiments, to deficiencies revealed in the expressive power of LCF as a logical language and to limitations on the deductive power of the machine implementation of the logic.

A dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Reproduced from best available copy.

ACKNOWLEDGEMENTS

I feel a great debt to John McCarthy, Robin Milner and Richard Weyhrauch who kindled my interest in the logical aspects of Computer Science. These people were also the main source of background and principal agents of stimulation for my research. In particular Robin Milner deserves a lot of credit for his vision in originating the LCF project.

Special thanks are due to Tony Hoare for his very helpful and detailed criticism of the first draft of this thesis.

I would also like to thank David Barstow for the help which made it possible for me to edit the dissertation and produce hard copy at Stanford while living in Edinburgh.

Finally, I would like to express my deep gratitude to my wife, Marie, for lots of assistance, patience and encouragement.

TABLE OF CONTENTS

SECTION or CHAPTER **PAGE**

1 Introduction 1

- 1.1 Summary 1
- 1.2 History of the 'LCF Project' 1
- 1.3 Past Work in Formal Semantics 4
- 1.4 A Treatment of Pure LISP 5
- 1.5 Correctness of a Compiler 7
- 1.6 Second Generation LCF 9
- 1.7 The Problem of Side-Effects 9

2 Pure LCF 11

- 2.1 Terms, Types and Domains: 11
  - 2.1.1 Syntax 11
  - 2.1.2 Semantics 12
  - 2.1.3 Strictness and Discreteness 13
  - 2.1.4 An Example 14
- 2.2 Formulae, Sentences and Proofs: 15
- 2.3 The Axioms and Rules of Inference. 16

2.4 Some Examples: 17
    2.4.1 $A \equiv B, B \equiv C \vdash A \equiv C$. 17
    2.4.2 $F \sqsubseteq G, A \sqsubseteq B \vdash F(A) \sqsubseteq G(B)$. 18

3 The LCF Proof Checking System 19
3.1 Proofs: 19
3.2 Formulae and Terms: 20
3.3 Using the LCF System: 23
    3.3.1 Substitution: 24
    3.3.2 Contradictions: 24
    3.3.3 Theorems: 25
    3.3.4 Simplification: 26
    3.3.5 Prefix Stripping: 29
3.4 Examples of LCF System Proofs: 29
    3.4.1 $A \sqsubseteq B, F \sqsubseteq G \vdash F(A) \sqsubseteq G(B)$. 29
    3.4.2 $(P \rightarrow \bot, \mathbb{F}) \equiv \mathbb{F} \vdash P \equiv \mathbb{F}$ 30
3.5 Concrete Representation: 32

4 The Mathematical Environment 33
4.1 Axiom Free Theorems in LCF: 34
4.2 Equality and Definedness: 35
4.3 Natural Numbers: 37

4.4 The Integers: 38

4.5 Integer Arithmetic: 40

4.6 A Theory of Lists: 41

5 Notation, Denotation and the Nature of LISP Expressions 44

5.1 Notation and Denotation: 44

5.2 Abstract Syntax 45

5.3 S-expressions: 46

5.4 LISP Expressions: 48

5.4.1 List Notation. 50

5.4.2 LISP Functions: 51

6 An Axiomatic Theory of Pure LISP 52

6.1 Extending the Environment for Names: 52

6.2 Axioms for Interpreting Pure LISP: 54

6.3 Discussion of the Axioms: 60

6.3.1 A Different 'evcon'. 60

6.3.2 'lisp' is not 'evalquote'. 61

6.3.3 Strictness of 'eval' and 'apply'. 61

6.3.4 Total Formality. 63

6.4 Theorems of Pure LISP: 63

7 Applications of the Theory of Pure LISP 67
- 7.1 The NULL Function: 69
- 7.2 The EQUAL Function: 71
- 7.3 The ASSOC Function: 73
- 7.4 Remarks: 74

8 The Correctness of an Interpreter 75
- 8.1 Meaning of PAIRLIS: 78
- 8.2 Important Lemmas: 79
- 8.3 Informal Proof of Interpreter Correctness: 81
- 8.4 Interpreter Correctness in LCF: 86
- 8.5 Partial Correctness: 87
- 8.6 Total Correctness: 90

9 Compiler Correctness (I) - Language Definitions 93
- 9.1 Extensions to the Environment: 93
- 9.2 LCom0 LISP: 95
  - 9.2.1 Informal Description 95
  - 9.2.2 Formal Description 96
  - 9.2.3 Theory of LCom0 LISP 100
  - 9.2.4 'BFD' - Basic Functions Defined 100
  - 9.2.5 Well-Formedness Predicate 102

9.3 LCom0 LAP - Informal Description: 102

9.4 LCom0 LAP - Formal Description: 106

9.4.1 States and functions on states: 106

9.4.2 LAP Functions and operations on them: 108

9.4.3 Interpreting LAP. 109

9.5 Towards a Theory of LAP. 111

10 Compiler Correctness (II) - Outline of a Proof 116

10.1 The Compiler: 116

10.1.1 Some Slight Changes 120

10.1.2 Predicate 'CFD' - Compiler Functions Defined 120

10.2 Meaning of the Compiler: 121

10.3 Properties of the Compiler Functions. 126

10.3.1 Totality 127

10.3.2 Completeness 128

10.3.3 Distribution of Labels 129

10.4 Statement of Correctness. 129

10.4.1 Correctness of the Compiling Algorithm 130

10.4.2 The Principal Lemma 131

10.4.3 Environment Correspondence 131

10.4.4 Second Level Subgoals 132

10.4.5 Attacking the Subgoals. 138

10.5 Feasibility of a Full Compiler Proof 139

11 Second generation LCF System 141

11.1 Prior Accomplishments 141

11.2 Proof Generation vs. Proof Checking 142

11.3 High Level Command Language 143

11.3.1 Data Types and Expressions 144

11.3.2 Control Structures 144

11.4 Revised Axiom Structure 145

11.5 Extending the Pure Logic 146

11.5.1 Derived Deduction Rules 147

11.6 Concrete Syntax 148

11.7 Extending Simplification 148

11.7.1 Inequalities 148

11.7.2 Split Level Simplification 149

11.7.3 n-time Simplification 149

11.7.4 Subgoals from Conditional Simplification 149

11.7.5 Case Analysis in Simplification 150

11.7.6 Simplifying Procedures 150

11.8 Types 150

11.9 Miscellaneous Improvements 152
11.9.1 Solving Equations 152
11.9.2 Definitional Facilities 153
11.9.3 Automatic Forward Reasoning 153
11.9.4 More Abbreviations 153

12 Conclusion 155

APPENDIX 1 - Theorems of LCom0 LISP 159

APPENDIX 2 - Yet Another LISP Subset 165

REFERENCES 169

LIST OF FIGURES

TITLE	PAGE
The Pure LISP Interpreter of McCarthy	55
The Definition of 'Eval'.	59
S-expression Form of the Interpreter.	76
S-expression Form (ctd).	77
Some Lemmas about SevlisB and SapplyB.	82
Some Lemmas about SevconB and SevalB.	83
Some Lemmas about eval, apply, evlis & evcon.	84
The Important Partial Correctness Subgoals.	89
Axioms for LCom0 LISP.	97
The Built-in Functions of LCom0 LISP.	98
Relationships Between 'eval', 'apply' etc.	99
Some Basic LISP Functions.	101
Well-Formedness of LISP expressions.	103
Partial Semantics of 8 Lap Instructions.	112
Partial Semantics of the CALL Instruction.	113
The LISP Functions that Make up LCom0.	117
LCom0 LISP Functions (ctd).	118
LCom0 LISP Functions (ctd).	119

'comp' - the meaning of 'COMP'. 122

Auxiliary Functions for 'comp'. 123

Auxiliary Functions for 'comp'. 124

Theorems Explicating 'compexp' and 'combool' 125

Subgoals Describing Effects of 'compexp'. 133

Subgoals Describing Effects of 'complis'. 134

Subgoals Describing Effects of 'comcond'. 135

Subgoal Describing Effects of 'combool'. 136

Subgoal Describing Effects of 'compandor'. 137

Axioms for Yet Another LISP. 166

Axioms for LISP (ctd). 167

Axioms for LISP (ctd). 168

CHAPTER 1

Introduction

1.1. Summary

This thesis discusses the application of LCF (Logic for Computable Functions) to the problem "Given a programming language, define precisely the semantics and develop a mathematical theory which is suitable for reasoning about programs of the language". It is primarily concerned with building an axiomatic theory of Pure LISP which can be used in the extraction of meanings of LISP functions. Particular functions discussed, in terms of correctness, are ones which perform interpretation and compilation of subsets of LISP. A principal aim of the investigation was an evaluation of both the expressive and the deductive power of LCF.

1.2. History of the 'LCF Project'

The starting point was an underground paper by Dana Scott [1] in 1969 describing a typed combinatory logic which was suitable for recursive function theory. Robin Milner, in 1971, replaced the combinators with typed lambda calculus and that logic will be referred to as Pure LCF. He also implemented a proof checker for this version of the logic and this program (later improved upon) is called the LCF System, or simply LCF. Milner described Pure LCF and the LCF System in [2] (including some examples of the use of the program), [3] is a user's manual for the system and [4] contains the only available technical discussion of the model theory of the logic. For the sake of self-containment, a short tutorial on Pure LCF is included in this report as Chapter 2 and because improvements to the LCF System are a major concern of this study, Chapter 3 is a brief description of the LCF System as it now exists (in fact an improvement on the version described in [3]).

Milner saw LCF as an excellent tool for the Mathematical Theory of Computation (MTC) and it is in this capacity that LCF has attracted some attention (within the field of computer science). The first chapters in the application of this tool to MTC problems were written by Milner and Weyhrauch (1972) with two documented experiments involving proofs of program properties. [5] discusses the proof of the correctness and termination of a simple program (for the factorial function) in a simple algebraic language defined by means of its abstract syntax. [6] reports on the development in the LCF System of a proof of the correctness of a simple compiling algorithm. That algorithm dealt with the abstract analytic syntax of the source language which featured the constructs of arithmetic expression (with binary operators and variables) with assignment, conditional, compound and 'while' statements. The target language was for a machine with an accumulator, a memory and a stack; it contained conditional and absolute jumps, load and fetch commands, labels and an instruction to apply arbitrary binary operators. Much of this experiment was concerned with the application of algebraic techniques to give structure to the proof.

Although the proofs in the Weyhrauch-Milner experiments were machine checked, it was expedient to assume many theorems from areas such as arithmetic and finite set theory rather than prove them in axiomatically based subtheories. The results in question were all considered 'intuitively obvious' but the practice allows errors to creep in. What was needed for succeeding experiments was a mathematical environment based on axiomatic treatments of the usual background areas such as arithmetic. A step in this direction was taken by Newey [7] who gave suitable developments of a basic propositional logic, natural numbers, arithmetic over the integers, lists and finite sets. The library of results obtained in that venture amounted to some 1000 theorems and was more comprehensive than our present needs require. We give, therefore, Chapter 4 as summary of of the parts of [7] that are relevant to giving the semantics of LISP.

Viewed in the light of this history, the formalisation of LISP semantics (in LCF) appears as another step in the application of LCF to the problems of MTC. In fact one of the main concerns in the experimental work is that it should inspire criticism of the current LCF system that can be translated into improvements to be realized in the next version.

1.3. Past Work in Formal Semantics

A survey on semantics of programming languages was given by J.W. de Bakker in [18]. Although it is getting old, we shall simply update it with pointers in the bibliography to more recent work by Burstall, Gordon, Hoare, Lauer, Manna, Waldinger and the Oxford school as well as the Milner-Weyhrauch work cited above. Of particular relevance is a short survey in [6] on compiler correctness.

There have not yet been any critical comparisons with previous formalisms but certain properties of LCF must be conceded to be big advantages. First it is based in logic and so it has deductive as well as expressive power (i.e. we can use it to reason about programs as well as define the semantics of languages). Second, it deals with functions (possibly partial) and functionals conveniently because of the lambda calculus base. Last, there are very good chances that automatic deduction will be moderately successful.

In terms of foundations for the present work, we follow the constructive approach that McCarthy has used but do it axiomatically in a logic as Burstall proposed. When we develop mathematical theories of a language we get theorems about the local effect of language features that rather resemble Hoare's rules. We also depend heavily on McCarthy's notion of abstract syntax as presented in [25] and [26].

Chapter 5 shows how we are able to factor syntax and semantics for LISP. The technique makes use of abstract syntax and functions for mapping between concrete text and abstract representations of programs and data. That chapter discusses the concepts of 'notation' and 'denotation' in relation to LISP.

1.4. A Treatment of Pure LISP

McCarthy presented Pure LISP in [15] but we take [12] to be the authorative reference since it is later (1962) and a touch smoother. Following his example we specify the language by means of an 'interpretive semantics' which uses association lists to bind values to variables. More precisely, taking both LISP data and functions to be S-expressions over a suitable set of names, a function is defined in LCF in such a way that it interprets source LISP expressions appropriately. Moreover, that function makes use of 'eval' and 'apply' functions which behave as the McCarthy Pure LISP functions of the same names. The LCF definitions of these functions together with the axioms which specify the notions of 'name' and

'S-expression' form a basis for a mathematical theory of Pure LISP. Chapter 6 presents the axioms and describes a rudimentary theory (a body of theorems) which will greatly ease the task of proving things about Pure LISP functions.

This semantics (or theory) was then used to prove that certain sample Pure LISP expressions denote the appropriate mappings on S-expressions. The particular functions were NULL, EQUAL and ASSOC. Chapter 7 discusses the proofs which were generated and checked using the LCF system since they illustrate some general techniques.

The examples culminate in a discussion of the correctness of the S-expression version of McCarthy's interpreter for Pure LISP which is written in Pure LISP itself. Actually, we will seek to establish the correctness of the S-expression form of 'eval' which we will call Seval. The property we want to prove is "For any A-list **al**, the function denoted by Seval via interpretation is 'eval' itself".

i.e. $\forall e\ a.\ \mathrm{apply}(\mathrm{Seval},\langle e\ a\rangle,\mathbf{al}) \equiv \mathrm{eval}(e,a)$

Chapter 8 addresses this problem and presents lemmas (proved with assistance of the LCF system) which show, in particular, that the functions 'eval' and $[\lambda e\ a.\ \mathrm{apply}(\mathrm{Seval},\langle e\ a\rangle,\mathbf{al})]$ satisfy almost identical recursive equations. These lemmas enable us to conclude in the metatheory of LCF that the functions are indeed the same. Reasoning within the logic it was possible to prove a sort of weak correctness:

$$\forall e\ a.\ \mathrm{eval}(e,a) \sqsubseteq \mathrm{apply}(\mathrm{Seval},\{e\ a\},\mathbf{al})$$

but the attempts to prove the other half of the above equality led to an identification of a deficiency in the LCF system. More specifically the other part of the proof would have required more space and time for computation than feasible.

1.5. Correctness of a Compiler

London in [13] gave a rather informal proof of the correctness of a certain compiler for a subset of LISP; LAP (a variety of PDP10 machine code) was the target language. This compiler, which is called LCom0, was written by McCarthy as a pedagogic device for a course at Stanford. Also, as mentioned before, Milner and Weyhrauch gave a formal proof of a minimal compiling algorithm using LCF. It was therefore clear that LCF was an appropriate vehicle for attempting the rigorous verification of compilers like LCom0.

Two chapters are devoted to a detailed study of the feasibility of establishing the correctness of LCom0 within the LCF system. The total task factors evenly to four subproblems. The first two are the axiomatisations of the two languages involved. The third is the extraction, from the S-expression version of the compiler, of its meaning function - 'the compiling algorithm'. The last is the establishment of the correctness of this compiling algorithm.

The treatment of the LISP subset parallels the work on Pure LISP in that axioms defining the language are expanded into a usable theory for the language by deriving theorems.

The meanings of those instructions that are generated by LCom0 are given in an abstract formalism which interprets the action of assembly code programs on machine states. The formalism is an abstraction in that no account is taken of store size, word size, the actual representation of S-expressions or garbage collection. As in the case of Pure LISP, certain handy lemmas are proved and described. This material takes us through Chapter 9.

Chapter 10 starts with the discussion of the extraction of the compiling algorithm from the S-expressions for LCom0. The same techniques illustrated in Chapter 7 are used although the larger S-expressions lead to correspondingly longer proofs.

The normal use of the compiler is to translate a 'program' of LISP functions into a program of LAP functions. We then say that a statement of compiler correctness is "in all such situations the the answer obtained by executing any LAP function must agree with the result of calling the corresponding LISP function with the same arguments".

Whereas our study of the other parts of the problem showed that attacks using LCF are quite feasible with the current LCF system, the proof of correctness of the compiling algorithm is much too long. In retrospect, this is not surprising since the compiler is an order of magnitude larger than the one Milner and Weyhrauch worked with and the languages are also more complicated.

Although the proof was not carried out we do discuss its structure and suggest in which directions the deductive power of the LCF system must be improved before the proof becomes feasible.

1.6. Second Generation LCF

Chapter 11 presents suggestions for the design of a new LCF system. The main design change is that the system should be two separate programs - a simple proof checker for a restricted form of LCF and an interactive proof generating program. There are also suggestions for making the input language to the system more 'high level'. A mechanism is presented for having a resticted class of derived deduction rules provable within LCF. Some attention is given to further extensions of simplification and some suggestions for new deduction mechanisms are examined.

1.7. The Problem of Side-Effects

Both subsets of LISP mentioned above contain just a few of the interesting features of 'practical' varieties of LISP. The most notable missing features are

SETQ's and PROG's. The second appendix gives another LCF interpretive semantics which can handle certain side-effect features of LISP - SETQs and the regular GENSYM device. It also deals with the PROG construct but still does not handle arrays or property lists and certainly not the distinction between the LISP 1.5 functions EQ and EQUAL. Again we deal with an idealisation of LISP which is not subject to recursion depth limits, finite arithmetic or bounded memory capacity.

CHAPTER 2

Pure LCF

In this short exposition of Scott's logic no justification of the semantics is given; the curious reader should consult [4].

2.1. **Terms, Types and Domains:**

2.1.1. Syntax

The **terms** of LCF are those of a typed λ-calculus with the addition of a least fixed point operator and certain constants; the two base types are called 'tr' (for truth values) and 'ind' (for individuals). All types other than 'tr' or 'ind' are derived from these two by a finite number of applications of the rule "If α and β denote types then so does $(\alpha\rightarrow\beta)$". With every term of the logic there is an associated type and we may postfix terms with their types so that, for example, **t**:β indicates that term **t** has type β. The syntax of LCF terms is then given by the productions:

```
              <term:β> = <identifier:β> | <application:β> |
                                 <conditional:β> | <λ-exprn:β> | <μ-exprn:β>

    where     <application:β2> = <term:(β1→β2)> ( <term:β1> )
```

<conditional:β> = <term:tr> $\rightarrow$ <term:β> , <term:β>

<λ-exprn:($\beta 1 \rightarrow \beta 2$)> = [$\lambda$ <identifier:$\beta 1$> . <term:$\beta 2$>]

<μ-exprn:β> = [μ <identifier:β> . <term:β>]

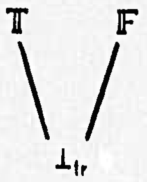
and where identifiers are defined in the usual way.

2.1.2. Semantics

Terms of type β denote objects in a **domain** D_β which is to be a partially ordered set (ordering relation $\sqsubseteq$) with every ascending chain having a least upper bound in D_β. Moreover, each D_β contains a minimum element denoted by the logical constant $\bot_\beta$; that is, $\bot_\beta \sqsubseteq x$ for all $x \in D_\beta$.

The interpretation intended for the relation '$\sqsubseteq$' in the various domains is that of relative definedness. That is, $x \sqsubseteq y$ is to be interpreted as saying y is at least as defined as (and consistent with) x. Hence we see that for a domain D_β, the interpretation of $\bot_\beta$ is that of the completely undefined element.

The base domain D_{tr} (the domain of truth values) contains precisely three elements ($\bot_{tr}$, $\mathbb{T}$, $\mathbb{F}$) in the fixed order given by the diagram:



The other base domain D_{ind} (the domain of individuals) is normally constrained by the addition of some non-logical axioms to characterise the non-functional data in a universe of discourse.

Finally, $D_{(\alpha\rightarrow\beta)}$ is the domain of continuous functions from D_α to D_β. A continuous function is one which preserves the least upper bounds of ascending chains. However, we shall never be using this notion explicitly so simply take it as fact that functions and functionals formed by all the term constructing mechanisms (presented above) are continuous. A property of these functions is that they are monotonic; i.e., if F is in $D_{(\alpha\rightarrow\beta)}$ and $x{:}\alpha \sqsubseteq y{:}\alpha$ then $F(x) \sqsubseteq F(y)$.

The interpretations of application and λ-abstraction are the usual ones. The term $S{:}tr \rightarrow T1{:}\beta, T2{:}\beta$ denotes $\perp_\beta$ or one of the two objects in D_β denoted by T1 and T2, according to whether S denotes $\perp_{tr}$, $\mathbb{T}$ or $\mathbb{F}$ respectively.

$[\mu f.S]$ should be interpreted as denoting the minimal fixed point of the function $[\lambda f.S]$. G is a fixed point of $[\mu f.g(f)]$ if G denotes the same function as $g(G)$; minimality is taken with respect to '$\sqsubseteq$'.

2.1.3. Strictness and Discreteness

A function $F_{(\alpha\rightarrow\beta)}$ is termed **strict** if the value of $F(\perp_\alpha)$ is $\perp_\beta$. A domain D_α is termed **discrete** or **flat** if for any x_α and y_α, $x_\alpha \sqsubseteq y_\alpha$ implies $x_\alpha \equiv y_\alpha$ or $x_\alpha \equiv \perp_\alpha$.

2.1.4. An Example

To illustrate these notions let us construct the factorial function in terms of arithmetic primitives. We imagine that we are given non-logical axioms which constrain the domain of individuals to contain a structure which looks like the natural numbers. So D_{ind} contains an individual which we call 0 and there is a successor function which generates all natural numbers by repeated application to 0 (1 is the successor of 0). We suppose Z is a predicate which is $\mathbb{T}$ on 0 and $\mathbb{F}$ on all other natural numbers. It is an easy exercise to use monotonicity to show $Z(\perp)$ must be $\perp$. (Note we are beginning to omit the mention of types when the information can be recovered from context.) We also make use of a predecessor function 'pred' and a two argument multiply function '*'.

$[\mu F.[\lambda x.\ Z(x) \rightarrow 1,\ *(x)(F(pred(x)))]]$, which we call 'fact', is an example of a term and contains instances of application, conditional expression, λ-abstraction and the minimal fixed point operation. It also involves bound variables ('x' and 'F'). This term denotes the least defined function which satisfies the recursive definition

$$F(x) \Leftarrow \text{if } x=0 \text{ then } 1 \text{ else } x*F(x-1) \quad .$$

The types of the various atoms are as follows:- '0', '1' and 'x' all have type ind; 'Z' has type (ind→tr); 'pred' and 'F' have type (ind→ind); '*' has type (ind→(ind→ind)). To illustrate why we are interested in least fixed points of functions note that the above recursive definition is satisfied by another function 'fact2' which agrees with 'fact' but gives zero on all negative numbers (assuming these are also in D_{ind}). It will be provable that fact $\sqsubseteq$ fact2 .

2.2. Formulae, Sentences and Proofs:

An **Atomic Well Formed Formula** (AWFF) has (for arbitrary type β) the form <term:β> $\sqsubseteq$ <term:β> . The symbol '$\sqsubseteq$' is of course identified with the ordering relation on D_β and so the interpretation of AWFFs is obvious.

A **Well Formed Formula** (WFF) is a **set** of (zero or more) AWFFs. WFFs are written as lists using comma as a separator. It follows from this definition that "$a \sqsubseteq b$, $c \sqsubseteq d$" is the same WFF as "$c \sqsubseteq d$, $a \sqsubseteq b$, $a \sqsubseteq b$" . A WFF is intended to denote the conjunction of its constituent AWFFs. Hence, the comma should be also interpreted as conjunction. We abbreviate "$s \sqsubseteq t$, $t \sqsubseteq s$" as "$s \equiv t$" .

An LCF **sentence** has the form P $\vdash$ Q where P and Q are WFFs. The 'turnstile' symbol should be interpreted as implication. If P is empty we omit it entirely.

Finally, a **proof** is a sequence of sentences with the property that each is either an instance of an axiom schema of Pure LCF or a deduction from previous sentences in the sequence using a rule of inference.

2.3. The Axioms and Rules of Inference.

We write $P\{s/x\}$ or $t\{s/x\}$ to mean the result of substituting s for all free occurrences of x in P or t, after first systematically changing bound identifiers in P or t so that no identifier free in s becomes bound by the substitution. Only λ and μ bind identifiers.

Inclusion axiom	$P \vdash Q$	(Q a subset of P)
Axioms for $\sqsubseteq$	$\vdash s \sqsubseteq s$ $s1 \sqsubseteq s2 \vdash t(s1) \sqsubseteq t(s2)$ $s1 \sqsubseteq s2, s2 \sqsubseteq s3 \vdash s1 \sqsubseteq s3$	 (Application) (Transitivity)
Axioms for $\bot$	$\vdash \bot \sqsubseteq s$ $\vdash \bot(s) \equiv \bot$	
Conditional axioms	$\vdash \bot \rightarrow s,t \equiv \bot$ $\vdash \mathbb{T} \rightarrow s,t \equiv s$ $\vdash \mathbb{F} \rightarrow s,t \equiv t$	
Conversion axioms	$\vdash [\lambda x.s](t) \equiv s\{t/x\}$ $\vdash [\lambda x.y(x)] \equiv y$	 (y distinct from x)
Fixed-point axiom	$\vdash [\mu x.s] \equiv s\{[\mu x.s]/x\}$	
Conjunction Rule	$\dfrac{P1 \vdash Q1 \qquad P1 \vdash Q2}{P1 \cup P2 \vdash Q1 \cup Q2}$	
Cut Rule	$\dfrac{P1 \vdash P2 \qquad P2 \vdash P3}{P1 \vdash P3}$	
Abstraction Rule	$\dfrac{P \vdash s \sqsubseteq t}{P \vdash [\lambda x.s] \sqsubseteq [\lambda x.t]}$	(x not free in P)

Cases Rule
$$\frac{P,s\equiv\mathbb{T} \vdash Q \qquad P,s\equiv\bot \vdash Q \qquad P,s\equiv\mathbb{F} \vdash Q}{P \vdash Q}$$

Induction Rule
$$\frac{P \vdash Q\{\bot/x\} \qquad P,Q \vdash Q\{t/x\}}{P \vdash Q\{[\mu x.t]/x\}} \qquad \text{(x not free in P)}$$

2.4. Some Examples:

2.4.1. $A\equiv B, B\equiv C \vdash A\equiv C$.

In this proof of an instance of the transitivity of '≡', note that the rules of Pure LCF are quite low level. The actual 'proof' is just the centre column of sentences and the justifications are for the benefit of the reader.

(a)	$A\equiv B \vdash A\sqsubseteq B$	by Inclusion Axiom;
(b)	$B\equiv C \vdash B\sqsubseteq C$	by Inclusion Axiom;
(c)	$A\equiv B, B\equiv C \vdash A\sqsubseteq B, B\sqsubseteq C$	by Conjunction,(a),(b);
(d)	$A\sqsubseteq B, B\sqsubseteq C \vdash A\sqsubseteq C$	by Transitivity Axiom;
(e)	$A\equiv B, B\equiv C \vdash A\sqsubseteq C$	by Cut,(c),(d);
(f)	$A\equiv B \vdash B\sqsubseteq A$	by Inclusion Axiom;
(g)	$B\equiv C \vdash C\sqsubseteq B$	by Inclusion Axiom;
(h)	$A\equiv B, B\equiv C \vdash C\sqsubseteq B, B\sqsubseteq A$	by Conjunction,(g),(f);
(j)	$C\sqsubseteq B, B\sqsubseteq A \vdash C\sqsubseteq A$	by Transitivity Axiom;
(k)	$A\equiv B, B\equiv C \vdash C\sqsubseteq A$	by Cut,(h),(j);
(l)	$A\equiv B, B\equiv C \vdash A\equiv C$	by Conjunction,(e),(k);

2.4.2. $F \sqsubseteq G,\ A \sqsubseteq B \vdash F(A) \sqsubseteq G(B)$.

Although this example is a trivial theorem of monotonicity it can be applied iteratively to get more complex theorems. Again the proof is quite tedious:

(a) $A \sqsubseteq B \ \vdash\ F(A) \sqsubseteq F(B)$ by Application Axiom;
(b) $F \sqsubseteq G \ \vdash\ [\lambda f.f(B)](F) \sqsubseteq [\lambda f.f(B)](G)$ by Application Axiom;
(c) $\vdash\ [\lambda f.f(B)](F) \equiv F(B)$ by a Conversion Axiom;
(d) $[\lambda f.f(B)](F) \equiv F(B) \ \vdash\ F(B) \sqsubseteq [\lambda f.f(B)](F)$ by Inclusion Axiom;
(e) $\vdash\ F(B) \sqsubseteq [\lambda f.f(B)](F)$ by Cut,(c),(d);
(f) $F \sqsubseteq G \ \vdash\ F(B) \sqsubseteq [\lambda f.f(B)](F),\ [\lambda f.f(B)](F) \sqsubseteq [\lambda f.f(B)](G)$
by Conjunction,(e),(b);
(g) $F(B) \sqsubseteq [\lambda f.f(B)](F),\ [\lambda f.f(B)](F) \sqsubseteq [\lambda f.f(B)](G) \ \vdash\ F(B) \sqsubseteq [\lambda f.f(B)](G)$
by Transitivity Axiom;
(h) $F \sqsubseteq G \ \vdash\ F(B) \sqsubseteq [\lambda f.f(B)](G)$ by Cut,(f),(g);

(j) $\vdash\ [\lambda f.f(B)](G) \equiv G(B)$ by a Conversion Axiom;
(k) $[\lambda f.f(B)](G) \equiv G(B) \ \vdash\ [\lambda f.f(B)](G) \sqsubseteq G(B)$ by Inclusion Axiom;
(l) $\vdash\ [\lambda f.f(B)](G) \sqsubseteq G(B)$ by Cut,(j),(k);
(m) $F \sqsubseteq G \ \vdash\ F(B) \sqsubseteq [\lambda f.f(B)](G),\ [\lambda f.f(B)](G) \sqsubseteq G(B)$
Conjunction,(h),(l);
(n) $F(B) \sqsubseteq [\lambda f.f(B)](G),\ [\lambda f.f(B)](G) \sqsubseteq G(B) \ \vdash\ F(B) \sqsubseteq G(B)$
by Transitivity Axiom;
(p) $F \sqsubseteq G \ \vdash\ F(B) \sqsubseteq G(B)$ by Cut,(m),(n);

(q) $A \sqsubseteq B,\ F \sqsubseteq G \ \vdash\ F(A) \sqsubseteq F(B),\ F(B) \sqsubseteq G(B)$ by Conjunction,(a),(p);
(r) $F(A) \sqsubseteq F(B),\ F(B) \sqsubseteq G(B) \ \vdash\ F(A) \sqsubseteq G(B)$ by Transitivity Axiom;
(s) $A \sqsubseteq B,\ F \sqsubseteq G \ \vdash\ F(A) \sqsubseteq G(B)$ by Cut,(q),(r);

CHAPTER 3

The LCF Proof Checking System

In this section we describe the computer program which aids in the generation of validated proofs in an enhanced version of LCF. Both the program and the enhanced logic are called simply "LCF" and ambiguities will be resolved by context. When we refer to the logic of Chapter 2 we shall always refer to it as Pure LCF.

3.1. **Proofs:**

A Pure LCF proof is a sequence of sentences subject to the condition that each of the sentences is an instance of one of the logical axiom schemas of Pure LCF or follows from previous sentences (in the sequence) by a rule of inference. A **proof** in the LCF implementation is a sequence of **'steps'** and a **step** is a four element list (**n**, **W**, **D**, **J**) where **n** is the step-number (an integer), **W** is an LCF WFF, **D** is a list of the dependencies of the step and **J** is the justification. Steps are numbered sequentially as they are generated and added to the end of the partial proof. The **dependencies** of a step are the step numbers of 'assumptions' on which the current step depends. The **justification** of a step indicates how the step was generated; it will include the name of the rule of inference employed and the previous steps that were used.

An assumption is a special step of the form (**n**, **W**, (**n**), (ASSUME **W**)). Note that the only dependency of an assumption is itself. Another special type of step is an axiom which has the form (**n**, **W**, (), (AXIOM **A**)) where **A** is the axiom name; note that axioms have no dependecies.

We now define the **sentence** denoted by a step (**n**, **W**, $(d_1,d_2,...d_m)$, **J**) to be $W_{d_1}, W_{d_2}, ... W_{d_m} \vdash \mathbf{W}$ where W_{d_i} is the WFF part of the line d_i (which will be an assumption). Thus, the step (**n**, **W**, (), (AXIOM **A**)) denotes the sentence $\vdash \mathbf{W}$ and the sentence denoted by the step (**n**, **W**, (**n**), (ASSUME **W**)) is clearly $\mathbf{W} \vdash \mathbf{W}$.

3.2. Formulae and Terms:

Not only do we have somewhat different notions of 'proof' in the pure logic and in implemented LCF (albeit there is a correspondence between them), but there are slight changes to the meanings of WFFs and terms.

First of all '≡' is not regarded as simply an abbreviation, but has a similar status to '⊑'. Thus, s≡t is regarded as an AWFF (as opposed to a WFF in Pure LCF) and there are deduction rules which deal with these 'equalities' (so called) rather than 'inequalities'. This change in approach is justifiable via the observation that '$\equiv$' is the much commoner relation and much easier to reason with. On the other hand, extra deduction rules are necessitated for conversion among the formulae $s\equiv t$, ($s\sqsubseteq t, t\sqsubseteq s$) and $t\equiv s$. The rules provided in the implementation are

$$\text{HALF}\quad \frac{s\equiv t}{s\sqsubseteq t}\qquad\qquad \text{SYM}\quad \frac{s\equiv t}{t\equiv s}\qquad\qquad \text{EQUIV}\quad \frac{s\sqsubseteq t, t\sqsubseteq s}{s\equiv t}$$

It should be noted that experience has (so far) indicated that these rules are rarely invoked (due in large measure to the rarity of the '$\sqsubseteq$' relation).

Next, also contrary to the definitions in Chapter 2, the WFFs are often regarded by the program more as lists than as sets. For example, $s\equiv t$ is not the same WFF as $s\equiv t, s\equiv t$. This is necessary to some extent since it is convenient to be able to talk about the n-th AWFF of a WFF but there is also some ugliness about the implementation in this respect.

In the current implementation there is no provision for talking about type information. (Hence there can be no type checking.)

Finally there are some very important abbreviations which are used by the program to make proofs more readable. These apply to both terms and AWFFS.

The following relate to terms:

i) $[\lambda a\ b.\ t]$ abbreviates $[\lambda a.\ [\lambda b.\ t]]$,

$[\lambda a\ b\ c.\ t]$ abbreviates $[\lambda a.\ [\lambda b.\ [\lambda c.\ t]]]$ etc.

ii) $s(t1,t2)$ abbreviates $s(t1)(t2)$,

$s(t1,t2,t3)$ abbreviates $s(t1)(t2)(t3)$ etc.

iii) if F is a function which normally takes 2 arguments then we may declare it infix and then we write $s\ F\ t$ for $F(s,t)$.

The following relate to AWFFS:

i) $\forall x.\ s \sqsubseteq t$ abbreviates $[\lambda x.s] \sqsubseteq [\lambda x.t]$ and

$\forall x.\ s \equiv t$ abbreviates $[\lambda x.s] \equiv [\lambda x.t]$;

The notation so introduced is very suggestive of its normal application: if we have $\forall x.\ s(x) \equiv t(x)$ then for all terms x we can deduce $s(x) \equiv t(x)$.

ii) $R \Rightarrow s \sqsubseteq t$ abbreviates $R \rightarrow s, \perp \sqsubseteq R \rightarrow t, \perp$ and

$R \Rightarrow s \equiv t$ abbreviates $R \rightarrow s, \perp \equiv R \rightarrow t, \perp$;

The structure abbreviated is an instance of a rather common device for relativising AWFFS. Noting that the sentence,

$$W \vdash R \Rightarrow s \sqsubseteq t$$

is equivalent to the other sentence,

$$W, R \equiv \mathbb{T} \vdash s \sqsubseteq t,$$

we see that the '$\Rightarrow$' connective corresponds to material implication.

3.3. Using the LCF System:

The LCF Implementation has really outgrown the name of 'proof checker'. Apart from the fact that a user rarely types a WFF (the information he gives is generally a sequence of commands that tell the machine HOW to generate the required sequence of steps), there are various mechanisms to help him interactively prove theorems in LCF. On the other hand, one couldn't be so bold to call it even an 'interactive' theorem prover, although this is a direction of future developments.

One of the most important aids to proof generation is the machinery that allows (even encourages) goal directed proving. A user may state target steps and attack them by indicating one of many tactics whereupon the program deduces appropriate subgoals and perhaps some relevant assumptions. Most of the tactics are the inverses of rules of inference since appropriate subgoals are ones which, if achieved, lead to the establishment of the goal by some rule of inference.

The inference rules of Pure LCF are rather basic and, in applications to MTC, too low level to be workable. However, the LCF system has five very important derived deduction mechanisms: substitution, contradiction, theorem use, simplification and prefix stripping.

3.3.1. Substitution:

Substitution is the implementation of three derived deduction rules of Pure LCF. The first two rules (following only from the CONV and ABSTR rules) are:

$$\frac{P \vdash t1 \sqsubseteq t2}{P \vdash s \sqsubseteq s\{t2/t1\}} \qquad \frac{P \vdash t1 \equiv t2}{P \vdash s \equiv s\{t2/t1\}}$$

and the third follows from these together with the TRANS rule (expanded to include the '≡' relation, of course):

$$\frac{P \vdash t1 \equiv t2 \ , \quad Q \vdash W}{P,Q \vdash \ W\{t2/t1\}}$$

There are the usual cautions about capture of bound variables.

3.3.2. Contradictions:

There is an inference rule which enables proofs by contradiction. We take it that the logic is consistent and so assuming that one's non-logical axioms are too, one can never prove sentences such as $\vdash \mathbb{T} \equiv \mathbb{F}$. Hence, given a step containing a 'contradictory' WFF (such as ' $\mathbb{T} \equiv \bot$ '), we should conclude that the dependencies are inconsistent. Now, given such a step with a 'contradictory' WFF (and dependencies D) in an LCF proof we could proceed to prove any other WFF with the same dependencies (D). (It is a nice exercise to show this.) The program recognizes the following four inequalities in D_{tr} as contradictions:

$$\mathbb{T} \sqsubseteq \bot \qquad \mathbb{F} \sqsubseteq \bot \qquad \mathbb{T} \sqsubseteq \mathbb{F} \qquad \mathbb{F} \sqsubseteq \mathbb{T}$$

as well as the six equalities between distinct members of D_{tr} (such as $\mathbb{T} \equiv \mathbb{F}$) and allows the user to prove any goal (i.e. make it a step) by claiming it follows from a contradiction. The resulting step will have the dependencies of the contradiction.

3.3.3. Theorems:

In the pure logic, a proof of a sentence, say $\vdash P \to \mathbb{T},\mathbb{F} \equiv P$, in no way constitutes a proof of any similar sentence (such as $\vdash Q \to \mathbb{T},\mathbb{F} \equiv Q$) which differs from the former only in the naming of free variables (which are not free in the axioms). However, it is clear that the ability to perform some renaming is absolutely necessary for a smooth system. In the LCF system such inferences are performed via the **theorem** mechanism.

At any point in a proof, a step may be given theorem status and the sentence that the step denotes acquires a name and is tagged with the names of the axioms that have been already introduced in the proof. There are, of course, two parts to a theorem: an antecedent WFF and a consequent WFF. The antecedent is the WFF denoting all dependencies and the consequent is the WFF part of the step. When the user desires to use a theorem, he may have the system change (throughout the theorem) any free variable (that is not free in any of the appropriate axioms) to any term (a process called **instantiation**), and by providing steps which when conjoined match the antecedent of the theorem, he may infer the consequent of the theorem. The dependencies part of the new step is the union of the dependencies of the steps used to match the antecedent. It should be noted that the user does not have to type any instantiations that the machine can deduce from the list of steps which must match the antecedent.

3.3.4. Simplification:

As an introduction to simplification, imagine we have three steps of a proof:

(n1)	$\forall a.\ F(a) \equiv a$	**(d1)** ,
.....	.....	
(n2)	$\forall b.\ G(b) \equiv H(b)$	**(d2)** ,
.....	.....	
(n3)	$M \equiv F(G(F(N)))$	**(d3)** .

It should be clear that we can proceed (using only features that have been discussed already) to a step which contains the WFF $M \equiv H(N)$ and has dependencies **d1** ∪ **d2** ∪ **d3**. We might easily proceed through intermediate steps which state $M \equiv F(G(N))$ and $M \equiv F(H(N))$. None of the proofs will be very short and the steps involved will probably help to obscure perhaps more interesting parts of the total proof.

In the LCF system sets of equalities (called 'simpsets') are maintained (by the user with help from the machine) to help in the automation of such sequences of simplifying substitutions. When the simplification mechanism is invoked, the item (which may be a goal, a step or a term) to be 'simplified' is scanned recursively (top down, left to right) for a subterm which 'matches' the left hand side of an equality in the current simpset. When such a match is found the right hand side of the equality is used to generate a replacement for the subterm. This simplification process continues until no subterms in the item can be matched to anything in the simpset.

When an AWFF from a step is 'put in' a simpset and it has the form $\forall x_1\ x_2 \ldots x_n.\ A \equiv B$, the 'universally quantified' variables (x_1, x_2, .. x_n) are replaced in $A \equiv B$ by 'metavariables' (α_1, α_2, ... α_n) and the new AWFF $A1 \equiv B1$ is added to the simpset. The raison d'etre for metavariables is that they will match any term. Thus, if the equality $\forall a.\ F(a) \equiv a$ is put in, 'a' becomes a metavariable and, for any term 't', occurences of 'F(t)' will be 'matched' and replaced by 't'.

Adding a step to the SIMPSET amounts to adding each of the equalities (AWFFs) that constitute the WFF. Steps in the simpset carry indication of their dependencies; and as a simplification proceeds a cumulative union is kept to which the dependencies of steps used are added; this union will be contained in the dependencies of any step generated as a result of the simplification.

Theorems with no antecedents go into simpsets just as steps do except that there are no dependencies and any free variables (that are not free in the appropriate axioms) are also made into metavariables.

Theorems with antecedents may be put in a simpset, and when they are used by the simplifier the phenomenon is known as conditional simplification. Suppose the theorem $F(a) \equiv G(b) \vdash H(a,b) \equiv \mathbb{T}$ (where variables 'a','b' are not free in the axioms) is put in the simpset. The 'a' and the 'b' become metavariables and the theorem is considered when a subterm (of a term being simplified) is of the form " $H(*,*)$ ". Suppose the term matched is $H(s,t)$. What the simplifer does, instead of simply replacing the term by $\mathbb{T}$ (as it would in the absence of an antecedent), is to attempt to verify the antecedents of the theorem by simplification. If the simplifier succeeds in checking the conditions of the theorem it performs the replacement called for by the consequent. There are depth bounds on the recursion in connection with this conditional simplification device.

Steps of the proof may also be conditional simplification rules. A WFF in a step such as $\forall x.\ p(x) \Rightarrow F(x) \equiv G(x)$, when added to the SIMPSET, is inserted in two ways - both using one metavariable 'β'. First way: the left hand side is "$p(\beta) \rightarrow F(\beta), \perp$" and the right hand side is "$p(\beta) \rightarrow G(\beta), \perp$" . Second way: the left and right sides are "$F(\beta)$" and "$G(\beta)$" respectively but there is also a condition to be checked - "$p(\beta) \equiv \mathbb{T}$".

3.3.5. Prefix Stripping:

When a GOAL is an AWFF with several prefixes ($\forall x.\ a(x) \Rightarrow \forall y\ z.\ A \equiv B$ has four prefixes) the natural way to attack it is by a series of abstractions (to remove outside universal quantifiers) and cases arguments (to remove relativisations) where two cases are trivial. This action can be performed in a single step by means of the PREF tactic. Abstractions are done automatically and a step is generated which corresponds to the nontrivial case of each relativisation. If the goal is actually achieved by the method then the cases steps are deleted.

For example, if the goal were $\forall x.\ A(x) \Rightarrow \forall y.\ B(y) \Rightarrow F(x,y) \equiv G(x,y)$ then the assumption steps generated and put in the simpset would be $A(x) \equiv \mathbb{T}$ and $B(y) \equiv \mathbb{T}$; the subgoal would be $F(x,y) \equiv G(x,y)$.

3.4. Examples of LCF System Proofs:

Before considering a significant example note that example 1 of Chapter 2 is a one step proof - namely, invocation of the 'Equivalence rule'.

3.4.1. $A \sqsubseteq B, F \sqsubseteq G \vdash F(A) \sqsubseteq G(B)$.

This is the other example of Chapter 2 and is much less painful using the LCF system. The text below is a conversation with the program. Each command to the system (the user's contribution to the interaction) follows a prompt of 5 stars and terminates with a semicolon. The integers are step numbers.

```
***** ASSUME A⊑B,F⊑G;
1  A ⊑ B  (1)
2  F ⊑ G  (2)

***** APPL F,1;
3  F(A) ⊑ F(B)  (1)

***** APPL 2,B;
4  F(B) ⊑ G(B)  (2)

***** TRANS 3,4;
5  F(A) ⊑ G(B)  (1 2)

***** THEOREM MONO1: -;

THEOREM MONO1: F(A) ⊑ G(B)
ASSUME
 A ⊑ B,
 F ⊑ G
```

3.4.2. $(P\to\bot,\mathbb{F})\equiv\mathbb{F} \vdash P\equiv\mathbb{F}$

This particular proof would already be something like fifty steps in Pure LCF. The material shown, this time, is not the whole record of the interaction but just the commands typed by the user and the proof that LCF constructs.

```
ASSUME  P→⊥,𝔽≡𝔽;
GOAL P≡𝔽;
TRY CASES P;
 TRY 1; SIMPL 1 BY -; QED;
 TRY 2; SIMPL 1 BY -; QED;
 TRY 3 SIMPL;
```

```
1   P→⊥,𝔽 ≡ 𝔽  (1) --- ASSUME.
 --------------------
 |TRY #1   P ≡ 𝔽      CASES P.
 |--------------------
 | |TRY #1#1   P ≡ 𝔽 : SASSUME   P ≡ 𝕋    .
 | |2   P ≡ 𝕋  (2) --- SASSUME.
 | |3   ⊥ ≡ 𝔽  (1 2) --- SIMPL 1 BY 2 .
 | --------------------
 | --------------------
 | |TRY #1#2   P ≡ 𝔽 : SASSUME   P ≡ ⊥    .
 | |4   P ≡ ⊥  (4) --- SASSUME.
 | |5   ⊥ ≡ 𝔽  (1 4) --- SIMPL 1 BY 4 .
 | --------------------
 | --------------------
 | |TRY #1#3   P ≡ 𝔽 : SASSUME   P ≡ 𝔽    .
 | |6   P ≡ 𝔽  (6) --- SASSUME.
 | --------------------
 |7   P ≡ 𝔽  (1) --- CASES P 3 5 6.
 --------------------
```

Notes:

i) The goal structure is reflected by the boxes.
The goal is printed at the top of the box.
The last line of the box is the established goal or a contradictory WFF.

ii) The command "TRY CASES P" caused the "P≡𝔽" goal to split to 3 subgoals:

```
P≡𝔽  SASSUME   P≡𝕋
P≡𝔽  SASSUME   P≡⊥
P≡𝔽  SASSUME   P≡𝔽
```

iii) A QED command, when the current step is a contradiction, renders the goal under trial established.

iv) A SASSUME(W) command causes WFF W to be added to the simpset after it is ASSUMEd.

3.5. **Concrete Representation:**

It is unusual for computer input character sets to contain many of the logical symbols that we need and, although the machine at the Stanford A.I. Project is exceptional in this respect, it is missing some symbols we have used. On that machine (where the LCF system lives) the following representations are used for characters which do not appear on the keyboard:

μ is represented by α
$\perp$ is represented by UU
$\sqsubseteq$ is represented by $\subset$
$\Rightarrow$ is represented by ::
$\mathbb{T}$ is represented by TT
$\mathbb{F}$ is represented by FF

The point of mentioning this matter is that the concrete representations of these characters have appeared in published works where publication language should have been used.

CHAPTER 4

The Mathematical Environment

We present here a brief account of those parts of [7] which are relevent to the LISP semantics experiments. That paper discusses the rigorous development of theories of propositional logic, integer arithmetic, lists and finite sets. It provides the axiomatic basis for a library of standard theorems from those various theories as well as a collection of results (not depending on axioms) which are useful when working with the LCF system.

In proving theorems about the meanings (and other properties) of various LISP functions, it is necessary to make use of a substantial number of results from arithmetic. Also, of course, the theory of lists is fundamental to the representation of LISP functions and the data they manipulate. The set theory of [7] was not required as background mathematics. Moreover, it was convenient to avoid using the treatment of propositional logic (thus space was saved since the theorems were not required).

We proceed, in this summary, by giving the axiomatic bases for each of the various aspects of the environment together with some indication of the scope and depth of the corresponding sections of the library of theorems. As an indication of the numbers of theorems involved, we note that the number given in [7] is about 1000 and about 400 of these were selected for use in the LISP experiments. As was anticipated, this body of theorems needed to be extended by the addition of some other useful lemmas. About 40 such extra results were added to the environment (all having very short proofs).

The domain of individuals (D_{ind}) is thought of as partitioned into subdomains which correspond to data types. These subdomains are characterised by type-predicates (functions of LCF type (ind→tr)). For example the predicate 'isint' (axiomatised below) gives $\mathbb{T}$ on individuals which are supposed to be integers and $\bot$ or $\mathbb{F}$ on all else in D_{ind}.

4.1. Axiom Free Theorems in LCF:

The theorems (or classes of theorems) in the following list depend on no (nonlogical) axioms. None is very deep but they find frequent use.

i) $\vdash [\lambda x.\bot] \equiv \bot$
ii) $\vdash \forall p.\ p \rightarrow \mathbb{T},\mathbb{F} \equiv p$
iii) $\vdash \forall p.\ p \rightarrow \bot,\bot \equiv \bot$

iv) $X \sqsubseteq \bot \ \vdash \ X \equiv \bot$
v) $F(X) \equiv \bot \ \vdash \ F(\bot) \equiv \bot$
vi) $P(\bot) \equiv \mathbb{T} \ \vdash \ P \equiv [\lambda x.\mathbb{T}]$
$P(\bot) \equiv \mathbb{F} \ \vdash \ P \equiv [\lambda x.\mathbb{F}]$
vii) $P(X) \equiv \mathbb{T},\ P(Y) \equiv \mathbb{F} \ \vdash \ P(\bot) \equiv \bot$

viii) $F1 \sqsubseteq F2,\ A1 \sqsubseteq A2 \vdash F1(A1) \sqsubseteq F2(A2)$
$F1 \sqsubseteq F2,\ A1 \sqsubseteq A2,\ B1 \sqsubseteq B2 \vdash F1(A1,B1) \sqsubseteq F2(A2,B2)$
and so on.
ix) $P \rightarrow \mathbb{T},\bot \equiv \mathbb{T} \vdash P \equiv \mathbb{T}$
$P \rightarrow \mathbb{F},\mathbb{T} \equiv \bot \vdash P \equiv \bot$
etc.
x) $P \rightarrow \mathbb{T},Q \equiv \mathbb{F} \vdash P \equiv \mathbb{F}$
$P \rightarrow Q,\mathbb{T} \equiv \mathbb{F} \vdash Q \equiv \mathbb{F}$
etc.
xi) $G \equiv F(G) \vdash [\mu g.F(g)] \sqsubseteq G$

xii) $P \rightarrow \mathbb{T},\mathbb{T} \equiv \mathbb{F} \vdash \mathbb{T} \equiv \mathbb{F}$
$P \rightarrow \mathbb{F},\bot \equiv \mathbb{T} \vdash \mathbb{T} \equiv \mathbb{F}$
etc.

Note that (i), (ii), (iii) are suitable for permanent SIMPSET residence and that (xii) is good for deriving contradictions.

4.2. Equality and Definedness:

We are easily able to axiomatise a sensible equality predicate (=) and a definedness predicate (∂). We want to call all individuals except $\bot$ **defined**; that is, if x is in D_{ind} then we want $\partial(x) \equiv \mathbb{T}$ if and only if x is **not** $\bot$. The desired equality predicate must be $\mathbb{T}$ or $\mathbb{F}$ on all pairs of defined elements of D_{ind}, must be reflexive on defined elements and must be such that $(x=y) \equiv \mathbb{T}$ indicates $x \equiv y$. In postulating such a two place predicate we make a commitment that D_{ind} should be discrete (flat).

We axiomatise '=' and define '∂' in terms of it as follows:

AXIOM EQ

$$\forall x.\ (x{=}x)\rightarrow x,\bot \equiv x$$
$$\forall x\ y.\ (x{=}y)\Rightarrow x\equiv y,$$
$$\forall x\ y.\ (x{=}x)\rightarrow((y{=}y)\rightarrow\mathbb{T},\bot),\bot \equiv (x{=}y)\rightarrow\mathbb{T},\bot$$
$$(\bot{=}\bot) \equiv \bot$$

$$\partial \equiv [\lambda x.\ x{=}x\ ]$$

We use the vertical bars (||) down the left hand edge of the page to indicate axioms.

As a technical aside to the critical reader, note that the fourth of these axioms is not necessary if we can talk about some element of D_{ind} other than $\bot$; in that case we can deduce '$(\bot{=}\bot)\equiv\bot$' by monotonicity.

Although $(X{=}Y)\equiv\mathbb{T} \vdash X\equiv Y$ is the fundamental property of the equality predicate, '=' should not be confused with '≡'; the latter is not a computable function.

Both of these functions (definedness and computable equality) have proved extremely useful and the following are the theorems (or groups of theorems) that are to be found in the environment (with comments):-

i)	$\vdash\ \partial(\bot) \equiv \bot$	(Strictness of ∂)
ii)	$\vdash\ \forall x.\ x{=}\bot \equiv \bot$ $\vdash\ \forall x.\ \bot{=}x \equiv \bot$	(Strictness of '=')
iii)	$\vdash\ \forall x.\ (x{=}x) \equiv \partial(x)$	(Reflexivity of '=')
iv)	$\partial(X)\equiv\bot\ \vdash\ X \equiv \bot$	(Totality of '∂')
v)	$X{=}Y\equiv\bot,\ \partial(X)\equiv\mathbb{T}\ \vdash\ Y\equiv\bot$ $X{=}Y\equiv\mathbb{F}\ \vdash\ \partial(X)\equiv\mathbb{T}$ etc.	(Totality of '=')

vi) $X=Y\equiv\mathbb{T} \;\vdash\; X\equiv Y$ (Conversion to '≡')
vii) $X\equiv Y,\ \partial(X)\equiv\mathbb{T} \;\vdash\; X=Y\equiv\mathbb{T}$ (Conversion from '≡')
viii) $\partial(X)\equiv\mathbb{T} \;\vdash\; X=X\equiv\mathbb{T}$ (Reflexivity again)
ix) $(X=Y)\equiv Tr \;\vdash\; (Y(=X)\equiv Tr$ (Commutativity of '=')
x) $\partial(X)\equiv\mathbb{T},\ X\sqsubseteq Y \;\vdash\; X\equiv Y$ (Discreteness of D_{ind})
xi) $F(X)=F(Y)\equiv\mathbb{F} \;\vdash\; X=Y\equiv\mathbb{F}$
$F(X)=F(Y)\equiv\mathbb{F} \;\vdash\; F(\bot)\equiv\bot$
$P(X)\equiv\mathbb{T},\ P(Y)\equiv\mathbb{F} \;\vdash\; X=Y\equiv\mathbb{F}$

xii) $\partial(X)\equiv\mathbb{F} \;\vdash\; \mathbb{T}\equiv\mathbb{F}$
xiii) $(X=X)\equiv\mathbb{F} \;\vdash\; \mathbb{T}\equiv\mathbb{F}$
xiv) $(X=Y)\equiv\bot,\ \partial(X)\equiv\mathbb{T},\ \partial(Y)\equiv\mathbb{T} \;\vdash\; \mathbb{T}\equiv\mathbb{F}$
xv) $(X=Y)\equiv\mathbb{F},\ X\sqsubseteq Y \;\vdash\; \mathbb{T}\equiv\mathbb{F}$

Note that the theorems suitable for permanent simpset residence form the first group (i-iii) and those which are contradiction oriented have also been grouped together (as xii-xv).

4.3. Natural Numbers:

Although the natural numbers are not used, as such, in the LISP experiments, and although the theorems concerning these objects have been removed from the environment as described in [7], the foundations for the construction of the integers is the axiomatisation of natural numbers. The interpretations intended for the constants (0, 1, Z, isnat, succ, pred) are the natural ones. Note that 'isnat' is a type predicate which gives $\mathbb{T}$ on natural numbers and $\bot$ on everything else.

AXIOM NN:
$Z \equiv [\lambda x.\ x=0]$
$Z(0) \equiv \mathbb{T}$
$\mathrm{isnat} \equiv [\mu F.\ [\lambda x.\ Z(x) \rightarrow \mathbb{T}, F(\mathrm{pred}(x))\ ]]$
$\forall x.\ \mathrm{isnat}(x) \Rightarrow Z(x) \rightarrow 0, \mathrm{succ}(\mathrm{pred}(x)) \equiv X$
$\forall x.\ \mathrm{isnat}(x) \Rightarrow Z(\mathrm{succ}(x)) \equiv \mathbb{F}$
$\forall x.\ \mathrm{isnat}(x) \Rightarrow \mathrm{pred}(\mathrm{succ}(x)) \equiv x$
$1 \equiv \mathrm{succ}(0)$
$2 \equiv \mathrm{succ}(1)$

Although this set of axioms is simply a building block (in the current context), we give a set of derivable theorems which correspond to the traditional Peano Postulates. This indicates that one should expect all the usual results of basic number theory to be provable.

$$\mathrm{isnat}(0) \equiv \mathbb{T}$$

$$\mathrm{isnat}(X) \equiv \mathbb{T} \ \vdash \ \mathrm{isnat}(\mathrm{succ}(X)) \equiv \mathbb{T}$$

$$\mathrm{isnat}(X) \equiv \mathbb{T} \ \vdash \ (\mathrm{succ}(X)=0) \equiv \mathbb{F}$$

$$\mathrm{isnat}(X) \equiv \mathbb{T},\ \mathrm{isnat}(Y) \equiv \mathbb{T},\ \mathrm{succ}(X) \equiv \mathrm{succ}(Y) \ \vdash \ X \equiv Y$$

$$\begin{aligned} &g(0) \equiv \mathbb{T},\ \forall x.\ \mathrm{isnat}(x) \Rightarrow g(x) \Rightarrow g(\mathrm{succ}(x)) \equiv \mathbb{T} \\ &\qquad\qquad \vdash \ \forall x.\ \mathrm{isnat}(x) \Rightarrow g(x) \equiv \mathbb{T} \end{aligned}$$

4.4. The Integers:

The following axioms specify more completely the functions 'pred' and 'succ' (see above) and introduce the functions 'mns', 'pos' and 'isint'.

AXIOM INT:

$\forall x.\ \mathrm{isnat}(x) \Rightarrow \mathrm{pos}(x) \equiv Z(x) \to \mathbb{F}, \mathbb{T}$
$\forall x.\ \mathrm{pos}(x) \Rightarrow \mathrm{isnat}(x) \equiv \mathbb{T}$
$\forall x.\ \mathrm{pos}(\mathrm{mns}(x)) \equiv \mathrm{pos}(x) \to \mathbb{F}, Z(x) \to \mathbb{F}, \mathbb{T}$
$\forall x.\ \mathrm{pos}(x) \to \mathbb{T}, \mathbb{T} \equiv \mathrm{isint}(x) \to \mathbb{T}, \perp$
$\forall x.\ \mathrm{isint}(x) \to \mathrm{mns}(\mathrm{mns}(x)), \mathrm{mns}(x) \equiv \mathrm{isint}(x) \to x, \perp$
$\forall x.\ \mathrm{succ}(x) \equiv \mathrm{mns}(\mathrm{pred}(\mathrm{mns}(x)))$
$\forall x.\ \mathrm{pred}(x) \equiv \mathrm{mns}(\mathrm{succ}(\mathrm{mns}(x)))$
$[\lambda x.\ \mathrm{isint}(x) \to \mathbb{T}, \mathbb{T}] \equiv \partial$

We first show the results of applying the various functions to the small integers (0,1,2) and to the undefined element of D_{ind}.

i) $\mathrm{isint}(0) \equiv \mathbb{T}$, $\mathrm{isint}(1) \equiv \mathbb{T}$, $\mathrm{isint}(2) \equiv \mathbb{T}$, $\mathrm{isint}(\perp) \equiv \perp$

ii) $\mathrm{pos}(0) \equiv \mathbb{F}$, $\mathrm{pos}(1) \equiv \mathbb{T}$, $\mathrm{pos}(2) \equiv \mathbb{T}$, $\mathrm{pos}(\perp) \equiv \perp$

iii) $Z(0) \equiv \mathbb{T}$, $Z(11) \equiv \mathbb{F}$, $Z(2) \equiv \mathbb{F}$, $Z(\perp) \equiv \perp$

iv) $\partial(0) \equiv \mathbb{T}$, $\partial(1) \equiv \mathbb{T}$, $\partial(2) \equiv \mathbb{T}$, $\partial(\perp) \equiv \perp$

v) $\mathrm{succ}(0) \equiv 1$, $\mathrm{succ}(1) \equiv 2$, $\mathrm{succ}(\perp) \equiv \perp$

vi) $\mathrm{pred}(1) \equiv 0$, $\mathrm{pred}(2) \equiv 1$, $\mathrm{pred}(\perp) \equiv \perp$

vii) $\mathrm{mns}(0) \equiv 0$, $\mathrm{mns}(\perp) \equiv \perp$

The derived theorems are too numerous to list but we now give some examples selected to give a flavor of them.

i) $\mathrm{isint}(X) \equiv \mathbb{F} \vdash \mathrm{pos}(X) \equiv \perp$
$\mathrm{isint}(X) \equiv \mathbb{F} \vdash \mathrm{succ}(X) \equiv \perp$

ii) $\vdash \forall x.\ \mathrm{isint}(\mathrm{succ}(x)) \equiv \mathrm{isint}(x) \to \mathbb{T}, \perp$
$\vdash \forall x.\ Z(\mathrm{mns}(x)) \equiv \mathrm{isint}(x) \to Z(x), \perp$
$\vdash \forall x.\ \mathrm{mns}(\mathrm{pred}(x)) \equiv \mathrm{succ}(\mathrm{mns}(x))$

iii) $\mathrm{pos}(X)\equiv\mathbb{T} \vdash \mathrm{isint}(X)\equiv\mathbb{T}$
$\mathrm{isint}(X)\equiv\mathbb{T} \vdash \partial(X)\equiv\mathbb{T}$
$\partial(\mathrm{succ}(X))\equiv\mathbb{T} \vdash \mathrm{isint}(X)\equiv\mathbb{T}$

iv) $\mathrm{isint}(X)\equiv\mathbb{T}, \mathrm{pos}(X)\equiv\perp \vdash \mathbb{T}\equiv\mathbb{F}$
$\mathrm{isint}(X)\equiv\mathbb{T}, Z(X)\equiv\perp \vdash \mathbb{T}\equiv\mathbb{F}$

4.5. Integer Arithmetic:

LCF is such that once we have axiomatised a structure then many of the functions we may be interested in can be written as terms of the logic. We give below definitions of the various operations of arithmetic that were appropriate to proving things about the LISP subsets that we are interested in.

AXIOM ARITH
$+ \equiv [\mu G.\ [\lambda x\ y.\ Z(y) \rightarrow (\mathrm{isint}(x)\rightarrow x,\perp),$
$\mathrm{pos}(y) \rightarrow G(\mathrm{succ}(x),\mathrm{pred}(y)),\ G(\mathrm{pred}(x),\mathrm{succ}(y))]]$

$- \equiv [\lambda x\ y.\ x+\mathrm{mns}(y)]$

$* \equiv [\mu G.\ [\lambda x\ y.\ Z(y) \rightarrow (\mathrm{isint}(x)\rightarrow 0,\perp),$
$\mathrm{pos}(y) \rightarrow G(x,\mathrm{pred}(y))+x,\ G(x,\mathrm{succ}(y))-x\ ]]$

$> \equiv [\lambda x\ y.\ \mathrm{pos}(x-y)]$
$\geq \equiv [\lambda x\ y.\ Z(x-y) \rightarrow \mathbb{T},\ \mathrm{pos}(x-y)]$

Many other useful and traditional arithmetic functions are defined in [7] including division, remainder-on-division and bounded-existential and bounded-universal quantifiers for integer predicates.

It is readily proved that all these functions are total over the integers but defined only on the integers; these facts find expression in many theorems in the environment. Apart from all the well-known basic properties of these functions (such as commutativity of '+' and '*' or the transitivity of '≥' and '>') being given, a large number of simple relations between 2 or 3 of the constants (⊥, 0, 1, succ, +, pred, -, *, ≥, >, mns) are given as theorems. In fact, the environment contains over 150 such theorems and there seems no way of categorising them so we can even list representative theorems. However, it has turned out that this library has been adequate to handle the modest requirements of the LISP project.

4.6. A Theory of Lists:

In [7] there is an extensive treatment of lists based on the axioms below. The treatment was substantially LISP-inspired and developed via a treatment of certain abstract objects that are similar to S-expressions. In that report they were called S-expressions but that has turned out to be a bad mistake for the current work so we will call them PONs (since a PON is either a Pair Or NIL). There is a pairing function '∘' (like CONS) and two selector functions ('hd' and 'tl' - like CAR and CDR) for analysing pairs. As in LISP an atom is anything that is not a pair and repeated selection in a PON eventually yields an atom.

AXIOM LIST

$\mathrm{ispon}(\bot) \equiv \bot$
$\mathrm{ispon}(\mathrm{NIL}) \equiv \mathbb{T}$
$\mathrm{null} \equiv [\lambda x.\ x \equiv \mathrm{NIL}]$
$\mathrm{atom} \equiv [\lambda x.\ \mathrm{ispon}(x) \rightarrow \mathrm{null}(x), \mathbb{T}]$
$\forall x.\ \mathrm{atom}(x) \Rightarrow \mathrm{hd}(x) \equiv \bot$
$\forall x.\ \mathrm{atom}(x) \Rightarrow \mathrm{tl}(x) \equiv \bot$
$\forall x\ y.\ \mathrm{hd}(x \cdot y) \equiv \partial(y) \rightarrow x, \bot$
$\forall x\ y.\ \mathrm{tl}(x \cdot y) \equiv \partial(x) \rightarrow y, \bot$
$\forall x.\ \mathrm{hd}(x) \cdot \mathrm{tl}(x) \equiv \mathrm{atom}(x) \rightarrow \bot, x$
$\partial \equiv [\mu G.[\lambda x. \mathrm{atom}(x) \rightarrow \mathbb{T}, G(\mathrm{hd}(x)) \rightarrow G(\mathrm{tl}(x)), \bot]]$
$\mathrm{islist} \equiv [\mu G.[\lambda x.\ \mathrm{null}(x) \rightarrow \mathbb{T}, \mathrm{atom}(x) \rightarrow \mathbb{F}, G(\mathrm{tl}(x))]]$

We first mention that all of the functions mentioned in the axioms are strict and that 'ispon', 'atom' & 'null' are total. We give just a few simple results of the theory (remembering that most of the theorems in the environment are quite simple):

i) $\partial(Y) \equiv \mathbb{T} \ \vdash\ \forall x.\ \mathrm{hd}(x \cdot Y) \equiv x$

ii) $\partial(X) \equiv \mathbb{T} \ \vdash\ \forall y.\ \mathrm{tl}(X \cdot y) \equiv y$

iii) $\partial(\mathrm{hd}(X)) \equiv \mathbb{T} \ \vdash\ \mathrm{atom}(X) \equiv \mathbb{F}$

iv) $\mathrm{atom}(X) \equiv \mathbb{F} \ \vdash\ \partial(\mathrm{hd}(X)) \equiv \mathbb{T}$

v) $\vdash\ \forall x.\ \partial(\mathrm{tl}(x)) \equiv \partial(\mathrm{hd}(x))$

vi) $\mathrm{null}(X \cdot Y) \equiv \mathbb{T} \ \vdash\ \mathbb{T} \equiv \mathbb{F}$

vii) $\mathrm{hd}(X) \equiv X \ \vdash\ X \equiv \bot$

viii) $\vdash\ \forall x\ y.\ \mathrm{islist}(x \cdot y) \equiv \partial(x) \rightarrow \mathrm{islist}(y), \bot$

ix) $$G(\mathrm{NIL})\equiv\mathbb{T},\ \forall x\ y.\ \partial(x)\Rightarrow \mathrm{islist}(y)\Rightarrow G(y)\Rightarrow G(x\cdot y)\equiv\mathbb{T}$$
$$\vdash\ \forall x.\ \mathrm{islist}(x)\Rightarrow G(x)\equiv\mathbb{T}$$

x) $$\forall x.\ \mathrm{atom}(x)\Rightarrow G(x)\equiv\mathbb{T},\ \ \forall x\ y.\ G(x)\Rightarrow G(y)\Rightarrow G(x\cdot y)\equiv\mathbb{T}$$
$$\vdash\ \forall x.\ \partial(x)\Rightarrow G(x)\equiv\mathbb{T}$$

We now come to present a selection of the various list operations that were defined in [7]; we define here only those operations that we require for the experiments in this thesis.

AXIOM LOP

$$\&\ \equiv [\mu G.[\lambda x\ y.\ \mathrm{null}(x)\rightarrow y,\ \mathrm{hd}(x)\cdot G(\mathrm{tl}(x),y)]]$$

$$\mathrm{mem} \equiv [\lambda x\ y.\ \partial(x)\rightarrow \mathrm{ORmap}(y,[\lambda z.x=z]),\bot]$$
$$\mathrm{ORmap} \equiv [\mu G.[\lambda x\ p.\ \mathrm{islist}(x)\rightarrow$$
$$(\mathrm{null}(x)\rightarrow\mathbb{F},\ p(\mathrm{hd}(x))\rightarrow\mathbb{T},\ G(\mathrm{tl}(x),p)),\bot]]$$

$$\mathrm{assoc} \equiv [\mu G.[\lambda x\ a.\ \partial(x)\rightarrow\ \mathrm{islist}(y)\rightarrow\ \mathrm{null}(y)\rightarrow\mathrm{NIL},$$
$$(x=\mathrm{hd}(\mathrm{hd}(y)))\rightarrow\mathrm{hd}(y),\ G(x,\mathrm{tl}(y)),\ \bot,\bot]]$$

$$\mathrm{length} \equiv [\mu G.[\lambda x.\ \mathrm{null}(x)\rightarrow 0,\ \mathrm{succ}(G(\mathrm{tl}(x)))]]$$

'&' is the append function for lists and 'mem' is membership in a list; 'assoc' and 'length' need no introduction. We do not give any properties of these functions but simply say that most of the results (about the functions) which were needed, were already available when required in the LISP experiments.

CHAPTER 5

Notation, Denotation and the Nature of LISP Expressions

5.1. Notation and Denotation:

We recall, here, the distinction between numbers and numerals. Numbers are abstract (mathematical) objects while numerals are expressions in certain languages; Numerals are used to denote numbers while numbers provide the interpretations for numerals. The common number/numeral confusion arises because of the usual identification of numbers with the numerals of the positional-notation decimal number system (actually a numeral system). Remember, every numeral **denotes** some number and is consequently **notation** for the number!

Chapter 4 described an environment within which the current experiments on a LISP semantics can be performed. Some very important classes of abstract objects are therein developed: - integers, lists and ordered pairs. A treatment of LISP must contain some discussion of notations for these abstract entities but we find our vocabulary is not rich enough: clearly 'list' corresponds to 'number' but we need a word to correspond to 'numeral'. We shall adopt the convention that when we have a name for a class of abstract objects we shall write it predominantly lower case and when we wish to discuss the class of expressions that represent the abstract objects we will use all capitals. For example an S-EXPRESSION will be an expression in a language that denotes a certain S-expression (an abstract object).

If we have a class Pqr of abstract objects and a class PQR of representations for elements of Pqr, then there is a semantic function, (call it **Den**) which maps expressions into the objects they denote. We will refer to **Den** as a **denotation** function. There are also functions which map each abstract object into an expression of the language we are using to discuss elements of Pqr; we call these functions **notation** functions.

Just as "02", "0002" and "2" are different notations for the same number, the LISP S-EXPRESSIONs "(A B)", "(A . (B))", "(A . (B . NIL))" and "(A . (B . ( )))" denote the same S-expression. The fact that systems of notation are often redundant in this way means that denotation functions are in general many to one. It is a fundamental property that if **N** is any notation function for Pqr then for all X in Pqr, **Den**(**N**(X)) ≡ X . Also, the function [λx. **N**(**Den**(x))] selects canonical representations.

5.2. Abstract Syntax

The term syntax usually refers to rules (perhaps phrased in BNF) which specify which strings of symbols are legal in some language and what the structures of the language are. McCarthy calls this 'concrete' syntax. 'Abstract' syntax also describes the structures of the language but without saying how the structures are represented by strings of symbols.

Abstract syntax comes in two flavours:- 'analytic' and 'synthetic'. Analytic abstract syntax makes use of **discriminators** such as 'issum' and 'isassignment' and also **selector** functions to access components of syntactic entities. Synthetic abstract syntax deals in **constructor** functions such as 'mksum' and 'mkassignment'.

Abstract syntax is no stranger to the LCF project - [5] and [6] depend on it. We now make the claim that defining denotation and notation functions (in LCF) in terms of McCarthy's notion of abstract syntax is quite straightforward. In the next section this assertion will be illustrated with definitions of such functions for S-expressions.

5.3. S-expressions:

As mentioned in Chapter 4, the notion of S-expression developed in [7] is unsatisfactory for our purposes. It is, therefore, part of the task of axiomatising subsets of LISP to define precisely what constitutes an S-expression. At this point, we can outline what makes one: a certain subset of the atoms of D_{ind} will be S-expressions and if we know X and Y are S-expressions then X·Y is one too. We cannot be specific about what the subset is, at this point, but it certainly will contain NIL and certain names or identifiers. Thus we are going to identify the LISP 'cons' function with the pairing function '·' that we know so much about. Similarly, we identify 'car' and 'cdr' with 'hd' and 'tl' respectively.

We are now in a position to exhibit denotation and notation functions for S-expressions. We use abstract syntax (both analytic and synthetic) in the definition. We suppose, for the sake of the example, that S-expressions are those individuals that satisfy the type-predicate:

$$\begin{aligned} \text{isSexprn} \equiv [\mu G.\ [\lambda x.\ &\text{isint}(x)\to\mathbb{T},\ \text{isname}(x)\to\mathbb{T},\ \text{null}(x)\to\mathbb{T},\\ &\text{atom}(x)\to\mathbb{F},\ G(\text{hd}(x))\to G(\text{tl}(x)),\ \mathbb{F}]]\ . \end{aligned}$$

We call the denotation function for S-EXPRESSIONS 'Sexprnof' and the notation function for S-expressions 'mkSEXPRN':

$$\begin{aligned} \text{Sexprnof} \equiv [\mu G.[\lambda X.&\text{isINTEGER}(X)\to\text{integerof}(X),\\ &\text{isNAME}(X)\to\text{nameof}(X),\\ &\text{isNIL}(X)\to\text{NIL},\\ &\text{isPAIR}(X)\to G(\text{leftof}(X))\cdot G(\text{rightof}(X)),\\ &\text{isLIST}(X)\to G(\text{firstof}(X))\cdot G(\text{restof}(X)),\\ &\bot\ ]] \end{aligned}$$

$$\begin{aligned} \text{mkSEXPRN} \equiv [\mu G.[\lambda x.&\text{isint}(x)\to\text{mkINTEGER}(x),\\ &\text{isname}(x)\to\text{mkNAME}(x),\\ &(x=\text{NIL})\to\text{mkNIL},\\ &\text{isSexprn}(x)\to\text{mkPAIR}(G(\text{hd}(x)),G(\text{tl}(x))),\\ &\bot\ ]] \end{aligned}$$

The functions 'isINTEGER', 'isNAME', 'isNIL', 'isPAIR' and 'isLIST' are analytic syntax discriminators; 'leftof', 'rightof', 'firstof' and 'lastof' are analytic syntax selector functions; 'mkPAIR' and 'mkNIL' are synthetic syntax constructor functions. Of course, we have just passed the buck since 'integerof' and 'nameof' are also denotation functions and 'mkINTEGER' and 'mkNAME' are notation functions.

If in this example we have appropriate results about the lower level functions such as

$$\forall x.\ \mathrm{isname}(x) \Rightarrow\ \mathrm{nameof}(\ \mathrm{mkNAME}(x)) \equiv x\ ,$$
$$\forall x.\ \mathrm{isINTEGER}(x) \Rightarrow\ \mathrm{isint}(\mathrm{integerof}(x)) \equiv \mathbb{T}\ ,$$
$$\forall x.\ \mathrm{isname}(x) \Rightarrow\ \mathrm{isINTEGER}(\ \mathrm{mkNAME}(x)) \equiv \mathbb{F}$$

then we are easily able to prove

$$\forall x.\ \mathrm{isSexprn}(x) \Rightarrow\ \mathrm{Sexprnof}(\mathrm{mkSEXPRN}(x)) \equiv x\ .$$

5.4. LISP Expressions:

Since this thesis is concerned with the semantics of the programming language LISP, we must inevitably describe what sort of mathematical object a LISP 'program' (or a LISP 'function') is. We must conclude that, because of the indistinguishability of program and data in LISP, all expressions in the language (whether they are intended for 'execution' or not) must have the same type; they must be members of D_{ind} . In fact all LISP functions, arguments and results will be S-EXPRESSIONs.

What we are looking for when we seek a semantics for LISP is a function (call it **LISP**) which maps S-EXPRESSIONS onto S-EXPRESSIONS in the same way as a LISP interpreter actually running in a machine. For example, **LISP** should map the S-EXPRESSION

"(CDR (CONS NIL (QUOTE X)))"

onto the S-EXPRESSION "X".

A necessary property of such a **LISP** function is that if S-EXPRESSIONS 'X' and 'Y' denote the same S-expression then the S-EXPRESSIONS '**LISP**(X)' and '**LISP**(Y)' must be the same.

We now point out a method of defining **LISP** indirectly which is very important to our work. What we do is to define an interpreting function 'lisp' which maps S-expressions onto S-expressions in the appropriate manner. Then by using denotation and notation functions (D and N) we can define **LISP** as a composition of functions:

$$\mathbf{LISP} = [\lambda X.\ N(\mathrm{lisp}(D(X)))]\ .$$

Note that the function we get depends on the particular choice of 'N' but this is as it should be. Because of the way we have defined **LISP** we have the following commutative diagram:

$$\begin{tikzcd}
\text{S-expressions} \arrow[r, "\text{lisp}"'] & \text{S-expressions} \arrow[d, "N"'] \\
\text{S-EXPRESSIONS} \arrow[u, "D"] \arrow[r, "\mathbf{LISP}"] & \text{S-EXPRESSIONS}
\end{tikzcd}$$

Now, if N_1 is any (other) notation function for S-expressions then we should be able to prove

$$\text{lisp} = [\lambda x.\ D(\mathbf{LISP}(N_1(x)))]$$

using the basic relationship between denotation and notation functions. This immediately suggests that the function 'lisp' is more fundamental than any particular **LISP** function we might have.

From this point on, therefore, we shall not be concerned with notation in general or S-EXPRESSIONs in particular; all discussion will centre round S-expressions.

5.4.1. List Notation.

We mention one point of notational convenience. In the LISP we all know and love, (A B C) can be thought of as an abbreviation of (A . (B . (C . NIL))). Just for the purposes of this document we shall use a similar abbreviation for lists but we use the distinctive brackets '{' and '}'. For example {A B C} is an abbreviation for $A\circ(B\circ(C\circ NIL))$ (i.e. denotes the list containing A, B, C). Note that (A B C) is not a term of LCF since the LCF system does not have a capability which allows introduction of abbreviations.

5.4.2. LISP Functions:

We have taken the position that LISP expressions, in general, and what are usually termed 'LISP Functions', in particular, are simply individuals. This raises the question "Do 'LISP Functions', such as

(LAMBDA (X) (CAR (CDR (CDR X))))

have any functional character whatsoever?" .

Answer: 'LISP Functions', although simply LISP data, **induce** functions under interpretation. Hence we may sometimes **identify** an S-expression with the LCF function that it induces under interpretation. For example, we will identify with the 'LISP Function' above, the LCF function:

[λy.lisp(((LAMBDA (X) (CAR (CDR (CDR X)))) y))] .

which will turn out to be simply the function [λy. hd(tl(tl(y)))] .

CHAPTER 6

An Axiomatic Theory of Pure LISP

6.1. Extending the Environment for Names:

Both data and programs in Pure LISP are S-expressions built from NIL and those atoms which are simply names (identifiers). The environment of Chapter 4 gives us some power to manipulate such S-expressions because of their structure but we need to augment these results so we can we can (logically) talk about the atoms in S-expressions. In fact, we must present axioms which further specify D_{ind} to contain names as well as integers etc. Not only do we want to talk about names in general but we want to introduce certain specific names such as 'T', 'LAMBDA' and 'CAR' .

The first four axioms for Pure LISP are then:

**AXIOM PL1:

$$\| \qquad \mathrm{isSexprn} \equiv [\mu F.[\lambda x.\ \mathrm{null}(x)\rightarrow\mathbb{T},\ \mathrm{isname}(x)\rightarrow\mathbb{T},\ \mathrm{atom}(x)\rightarrow\mathbb{F}, G(\mathrm{hd}(x))\rightarrow G(\mathrm{tl}(x)),\mathbb{F}]]$$

**AXIOM PL2:

$$\| \qquad \forall x.\ \mathrm{isname}(x) \rightarrow \mathrm{isint}(x) \rightarrow \bot,\ \mathrm{atom}(x) \rightarrow x,\ \bot,\ \mathrm{isint}(x) \rightarrow \mathrm{atom}(x) \rightarrow x,\ \bot,\ x \equiv x$$

**AXIOM PL3:

$$\| \qquad \forall x.\ \partial(\mathrm{discr}(x)) \Rightarrow \mathrm{isname}(x) \equiv \mathbb{T}$$

PL1 simply expresses in LCF the definition of S-expressions for Pure LISP which was given in plain language above. Then PL2 further specifies the structure of the domain of individuals (D_{ind}) as being partioned by the name and integer type-predicates. Looking at the consequences of these two axioms we see

$$\begin{array}{l} \vdash \ \text{isname}(\bot) \equiv \bot \\ \text{isname}(X) \equiv \bot \ \vdash \ X \equiv \bot \\ \text{isname}(X) \equiv \mathbb{T} \ \vdash \ \text{atom}(X) \equiv \mathbb{T} \\ \text{isint}(X) \equiv \mathbb{T} \ \vdash \ \text{atom}(X) \equiv \mathbb{T} \end{array}$$

as well as the fact that names, integers and pairs (non-atoms) are all distinct. Finally, PL3 introduces 'discr' which maps names onto integers and is the basis of a compact way of introducing specific names; we will use it is a discriminating function to give a total ordering for names (although this fact is not contained in the axiom). To illustrate its use we just proceed with the axioms for Pure LISP, giving the one which introduces the 'reserved words'.

**AXIOM PL4:

$$\begin{array}{l} \text{discr(LAMBDA)} > \text{discr(LABEL)} \equiv \mathbb{T}, \\ \text{discr(LABEL)} > \text{discr(QUOTE)} \equiv \mathbb{T}, \\ \text{discr(QUOTE)} > \text{discr(ATOM)} \equiv \mathbb{T}, \\ \text{discr(ATOM)} > \text{discr(COND)} \equiv \mathbb{T}, \\ \text{discr(COND)} > \text{discr(CONS)} \equiv \mathbb{T}, \\ \text{discr(CONS)} > \text{discr(CAR)} \equiv \mathbb{T}, \\ \text{discr(CAR)} > \text{discr(CDR)} \equiv \mathbb{T}, \\ \text{discr(CDR)} > \text{discr(EQ)} \equiv \mathbb{T}, \\ \text{discr(EQ)} > \text{discr(F)} \equiv \mathbb{T}, \\ \text{discr(F)} > \text{discr(T)} \equiv \mathbb{T} \end{array}$$

When using the LCF system to do the proofs discussed, we decorated the specific names (QUOTE, CAR etc.) with a leading underbar. Underbar in an identifier indicates that the atom is a constant name in D_{ind} and mentioned in the axioms. However in this report we will simply write _CAR as CAR .

It is a trivial exercise to show that each of these reserved words is a name (satisfies the 'isname' type-predicate). Furthermore, using the transitivity of '>' we can easily show that distinct names are unequal. For example, we can derive

$\mathrm{discr(CAR)} > \mathrm{discr(T)} \equiv \mathbb{T}$ and hence $\mathrm{CAR}=\mathrm{T} \equiv \mathbb{F}$.

6.2. **Axioms for Interpreting Pure LISP:**

As has already been inferred, we will be defining in this section, a function 'lisp' which 'interprets' S-expressions in the appropriate manner. For example, we wish the function to satisfy the equations

lisp({QUOTE T}) ≡ T

lisp({{LAMBDA {X} {CONS X {QUOTE F}}} {QUOTE T}}) ≡ T • F

where, of course, X is a name.

[12] contains, in order to be precise about the meaning of the language, an interpreter for Pure LISP. That interpreter, which is written in Pure LISP, and which we reproduce in Figure 6.1 (next page), is a collection of mutually recursive functions, the most important of which are 'eval' and 'apply'. 'eval' is a function of

```
apply[fn;x;a] =
     [atom[fn] → [eq[fn;CAR] → caar[x];
                  eq[fn;CDR] → cdar[x];
                  eq[fn;CONS] → cons[car[x];cadr[x]];
                  eq[fn;ATOM] → atom[car[x]];
                  eq[fn;EQ] → eq[car[x];cadr[x]];
                  T → apply[eval[fn;a];x;a]];
      eq[car[fn];LAMBDA] →
                 eval[caddr[fn];pairlis[cadr[fn];x;a]];
      eq[car[fn];LABEL] → apply[caddr[fn];x;
                       cons[cons[cadr[fn];caddr[fn]];a]]]

eval[e;a] =
     [atom[e] → cdr[assoc[e;a]];
      atom[car[e]] → [eq[car[e];QUOTE] → cadr[e];
                      eq[car[e];COND] → evcon[cdr[e];a];
                      T → apply[car[e];evlis[cdr[e];a];a]];
      T → apply[car[e];evlis[cdr[e];a];a]]

evcon[c;a] = [eval[caar[c];a] → eval[cadar[e];a];
              T → evcon[cdr[c];a]]

evlis[m,a] = [null[m] → NIL;
             T → cons[eval[car[m];a];evlis[cdr[m];a]]]

pairlis[x;y;a] = [null[x] → a;
                 T → cons[cons[car[x];car[y]];
                          pairlis[cdr[x];cdr[y];a]]]

assoc[x;a] = [equal[caar[a];x] → car[a];
              T → assoc[x;cdr[a]]]
```

Figure 6.1 - The Pure LISP Interpreter of McCarthy.

two arguments:- a LISP expression and an association list which is used to hold the bindings of variables. 'eval(**E**, **A**)' gives the LISP interpretation (the evaluation) of the expression **E** using **A** to get the values of variables. Similarly, 'apply(**F**, **L**, **A**)' applies the LISP function **F** to the list **L** of arguments again using **A** to bind values to variables.

Now in developing a new definition of Pure LISP, we do it in a way that corresponds as closely as possible to the McCarthy interpreter. In particular we will have LCF functions 'eval', 'apply', 'evlis', 'evcon' etc., each with almost the same structure as the LISP function of the same name.

'eval' is the most basic of the various functions we propose since we are able to define all the others in terms of it:

```
**AXIOM PL5:
            lisp ≡ [λe. eval(e,NIL) ]

**AXIOM PL6:
            apply ≡ [μG. [λfn x a. ∂(x)→ islist(a)→
                            (fn=CAR)→ hd(hd(x)),
                            (fn=CDR)→ tl(hd(x)),
                           (fn=CONS)→ hd(x)•hd(tl(x)),
                           (fn=ATOM)→ atom(hd(x)) → T, F,
                             (fn=EQ)→ [λx y.atom(x)→atom(y)→(x=y),⊥,⊥]
                                                         (hd(x),hd(tl(x)))→T,F,
                            atom(fn)→ G(eval(fn,a),x,a),
                    (hd(fn)=LAMBDA)→ eval(hd(tl(tl(fn)))), pairlis(hd(tl(fn)),x,a)),
                      (hd(fn)=LABEL)→ G(hd(tl(tl(fn))),x,((hd(tl(fn))•hd(tl(tl(fn))))•a)),
                                                               ⊥, ⊥, ⊥ ]]
```

**AXIOM PL7:

$$\begin{aligned}\mathrm{evcon} \equiv [\mu G.[\lambda c\ a.\ & (\mathrm{eval}(\mathrm{hd}(\mathrm{hd}(c)),a)=T) \rightarrow \mathrm{eval}(\mathrm{hd}(\mathrm{tl}(\mathrm{hd}(c))),a),\\ & (\mathrm{eval}(\mathrm{hd}(\mathrm{hd}(c)),a)=F) \rightarrow G(\mathrm{tl}(c),a)\ ,\ \bot]]\end{aligned}$$

**AXIOM PL8:

$$\mathrm{evlis} \equiv [\mu G.[\lambda m\ a.\mathrm{null}(m)\rightarrow \mathrm{NIL},\mathrm{eval}(\mathrm{hd}(m),a)\circ G(\mathrm{tl}(m),a)]]$$

**AXIOM PL9:

$$\begin{aligned}\mathrm{pairlis} \equiv [\mu G.[\lambda x\ y\ a.\ & \mathrm{null}(x)\rightarrow a,\\ & (\mathrm{hd}(x)\circ\mathrm{hd}(y)) \circ G(\mathrm{tl}(x),\mathrm{tl}(y),a)]]\end{aligned}$$

It remains only to define 'eval'. Inspired by the interpreter we want 'eval' to satisfy the equation

$$\begin{aligned}\mathrm{eval} \equiv [\lambda e\ a.\ & \mathrm{atom}(e) \rightarrow \mathrm{tl}(\mathrm{assoc}(e,a)),\\ & \mathrm{hd}(e)=\mathrm{QUOTE}) \rightarrow \mathrm{hd}(\mathrm{tl}(e)),\\ & (\mathrm{hd}(e)=\mathrm{COND}) \rightarrow \mathrm{evcon}(\mathrm{tl}(e),a),\\ & \qquad \mathrm{apply}(\mathrm{hd}(e),\mathrm{evlis}(\mathrm{tl}(e),a),a)]\ .\end{aligned}$$

Now this equation is not satisfactory as a definition since it contains references on the right to functions which depend on 'eval'; if we adopted this we would not have a set of definitions but a set of mutually recursive equations. Worse yet, this set of simultaneous equations, although consistent, does not specify the functions adequately. An example will show this: Consider the computation of

	eval({G}, {G∘{LAMBDA NIL {G}}})
through	apply(G, NIL, {G∘{LAMBDA ...}})
and	apply({LAMBDA NIL {G}},NIL,{G∘{LAMBDA...}})
back to	eval({G}, {G∘{LAMBDA NIL {G}}}) !!!

It is not inconsistent with the above equation, then, to assert, for example 'eval((G), (G·(LAMBDA NIL (G))) ≡ T' . We actually want our definition of 'eval' to specify the results of all computations.

The solution is clear, we take the definitions of 'evcon', 'apply' and 'evlis' and substitute them in the above equation for 'eval'; we then take the fixed point of the right hand side; lastly we add a leading condition to ensure strictness. We present the resulting axiom (PL10) as Figure 6.2 (next page). With this definition of 'eval' we get as a theorem

```
eval ≡ [λe a. islist(a)→
              (atom(e) → tl(assoc(e,a))),
              ((hd(e)=QUOTE) → hd(tl(e)),
               (hd(e)=COND) → evcon(tl(e),a),
                  apply( hd(e), evlis(tl(e),a), a)), ⊥]
```

A noteworthy technique for working in LCF was just used but the following abstract example will illustrate it better since it has less irrelevant detail; we suppose two functions (L,M) satisfy the equations:

$$L \equiv P(L,M) \quad \text{and} \quad M \equiv Q(L,M)$$

The MUTUAL least fixed points for L and M are given by the definitions:

$$L \equiv [\mu F.\ P(F,[\mu G.Q(F,G)])] \quad \text{and} \quad M \equiv [\mu G.\ Q(L,G)]\ .$$

Similarly, supposing three functions (L,M,N) satisfy the equations:

$$L \equiv P(L,M,N) \qquad M \equiv Q(L,M,N) \qquad N \equiv R(L,M,N)$$

the MUTUAL least fixed points for L,M,N are given by the quite lengthy definitions:

**AXIOM PL10:

```
eval ≡ [μB.[λe a.
    islist(a)→
       (atom(e) → tl(assoc(e,a)),
       hd(e)=QUOTE → hd(tl(e)),
       hd(e)=COND →
          [μG.[λc a. (B(hd(hd(c)),a)=T) → B(hd(tl(hd(c))),a),
              (B(hd(hd(c)),a)=F) → G(tl(c),a),⊥]](tl(e),a),

       [μG.[λfn x a. a(x)→ islist(a)→
              (fn=CAR) → hd(hd(x)),
              (fn=CDR) → tl(hd(x)),
            (fn=CONS) → hd(x)·hd(tl(x)),
            (fn=ATOM) → atom(hd(x)) → T, F,
               (fn=EQ) → [λx y.atom(x)→atom(y)→(x=y),
                               ⊥,⊥](hd(x),hd(tl(x)))→ T,F,
              atom(fn) → G(B(fn,a),x,a),
     (hd(fn)=LAMBDA) → B(hd(tl(tl(fn))), pairlis(hd(tl(fn)),x,a)),
       (hd(fn)=LABEL) → G(hd(tl(tl(fn))),x,
                          ((hd(tl(fn))·hd(tl(tl(fn))))·a)),⊥,⊥,⊥]]

                (hd(e),
                 [μG.[λm a. null(m)→NIL,
                         B(hd(m),a)·G(tl(m),a)]](tl(e),a),
                 a)), ⊥]
```

Figure 6.2 - The Definition of 'Eval'.

$$L \equiv [\mu F.\ P(F, [\mu G.\ Q(F,G,[\mu H. R(F,G,H)]],[\mu H.\ R(F,[\mu G. Q(F,G,H],H)]$$
$$M \equiv [\mu G.\ Q(L,G,[\mu H. R(L,G,H)]]$$
$$N \equiv [\mu H.\ R(L,M,H)]$$

6.3. **Discussion of the Axioms:**

6.3.1. A Different 'evcon'.

Because we have modelled the above definitions on McCarthy's interpreter, an actual difference in the semantics is accented - a difference in the actions of the two functions 'evcon'. That there is discrepancy is illustrated by the example:

```
eval( {COND { {QUOTE X} {QUOTE X} }
              { {QUOTE T} {QUOTE T} } }, NIL)
```

In our semantics this term is ⊥ whereas the old interpreter will yield the answer T. We feel justified in making this small change since it seems that the action of McCarthy's interpreter (in this case) is at variance with the natural language description of Pure LISP. We quote from [12] the definition of conditional expression:

> " A conditional expression has the following form:
> $$[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \ldots ; p_n \rightarrow e_n],$$
> where each p_i is an expression whose value may be truth or falsity, and each e_i is an expression. "

6.3.2. 'lisp' is not 'evalquote'.

[12] presents the top level of the Pure LISP interpreter to be the 'evalquote' function which corresponds to

$$[\lambda fn\ x.\ \mathrm{apply}(fn,x,\mathrm{NIL})]\ .$$

We could also have defined 'lisp' to be that term, but have chosen instead to follow the example of the usual LISP systems which use 'eval' as the 'top level'.

6.3.3. Strictness of 'eval' and 'apply'.

Next note that the definitions of 'eval' and 'apply' have the following structure:

$$\begin{array}{l} \mathrm{eval} \equiv [\mu F.[\lambda e\ a.\ \mathrm{islist}(a)\rightarrow(\mathrm{atom}(e)\rightarrow(..),(..)),\bot]] \\ \mathrm{apply} \equiv [\mu G.[\lambda fn\ x\ a.\ \partial(x)\rightarrow \mathrm{islist}(a)\rightarrow \\ \qquad\qquad ((fn=\mathrm{CAR})\rightarrow(..),(..)),\bot,\bot]]\ . \end{array}$$

The main point of the 'islist(a)' and '∂(x)' conditions is to ensure that each of these two functions is strict in each argument position. Of course, '∂(a)' would have guaranteed strictness equally well as 'islist(a)' but the latter was chosen for imagined technical convenience: we are only interested in the function when the last argument is an association list so it might as well be undefined if that argument is not even a list. In retrospect it would be preferable to replace 'islist(a)' by '∂(a)' in both definitions since some theorems are more compactly stated and many proofs become easier.

Actually, strictness in the last argument position for 'apply' is not essential but strictness in the second argument positions of both 'eval' and 'apply' is required to prevent counter-intuitive results. The following examples illustrate this fact:

i) lisp({LAMBDA {X} {QUOTE T}}·NIL) .

This term computes to

apply({LAMBDA {X} {QUOTE T}},NIL,NIL)

and then to

eval({QUOTE T},⊥) .

Depending on whether we have the 'islist(a)' condition in the definition of 'eval' or not this further computes to ⊥ or T respectively. Now ⊥ is the appropriate answer since a disaster occured during the computation. We would expect mechanical computation (as with an interpreter) of this example to FAIL at the point where 'hd(NIL)' is required.

ii) lisp({{LAMBDA NIL {QUOTE T}}
{ {LABEL N {LAMBDA NIL {N}}} NIL}})

Noting that

eval({{LABEL N {LAMBDA NIL {N}}} NIL},NIL) ≡ ⊥

This term computes to

apply({LAMBDA NIL {QUOTE T}}, ⊥, NIL) .

Then, depending on whether the '$\partial(x)$' condition is in the definition of 'apply' or not, we get, as the answer, either ⊥ or T . Again the latter answer is counter-intuitive since mechanical interpretation (using McCarthy's model) of

```
eval( ((LABEL N (LAMBDA NIL (N))) NIL),NIL)
```

would go on forever.

6.3.4. Total Formality.

Since the meaning of Pure LISP is embodied in the function 'lisp' in the axiomatic setting we have provided, we have succeeded in giving a completely formal specification of the language. Contrast this with the method of [12] where Pure LISP is first described in plain language and then this definition is 'tightened up' by the presentation of an interpreter. Note that this interpreter is not a definition of the language since it is only meaningful in the context of the accompanying natural language description.

6.4. Theorems of Pure LISP:

Having the definitions for 'lisp' and the auxiliary functions is barely half the job of constructing a 'theory of Pure LISP' that can be applied to proofs of correctness of programs. We now need to develop a body of theorems which we can expect will facilitate such applications. Presented in this section is such a collection of lemmas giving properties of the functions 'eval' etc. and giving the results of these functions in special cases. Most of the theorems are suitable for inclusion in a SIMPSET.

We start by presenting some lemmas for the functions 'eval', 'evlis' and 'pairlis'. These functions are strict in almost all argument positions. Where appropriate the strictness results such as $\vdash \forall x.\mathrm{evlis}(\bot,x) \equiv \bot$ were proved although we do not list them. More interesting are the following:

i) $\vdash \forall x\ y\ a.\ \mathrm{evcon}(\{\{\mathrm{QUOTE}\ T\}\ x\}\cdot y,a) \equiv \partial(y)\rightarrow \mathrm{eval}(x,a),\bot$

ii)
$$\begin{aligned}\partial(X)\equiv\mathbb{T},\ \partial(Y)\equiv\mathbb{T} \vdash\ & \forall w\ a.\ \mathrm{evcon}(\{w\ X\}\cdot Y,\ a)\\ &\equiv (\mathrm{eval}(w,a)=T)\rightarrow \mathrm{eval}(X,a),\\ &\quad (\mathrm{eval}(w,a)=F)\rightarrow \mathrm{evcon}(Y,a),\bot\end{aligned}$$

iii)
$$\begin{aligned}&\vdash \forall a.\ \mathrm{evlis}(\mathrm{NIL},a) \equiv \mathrm{NIL}\\ &\vdash \forall x\ a.\ \mathrm{evlis}(\{x\},a) \equiv \{\mathrm{eval}(x,a)\}\\ &\vdash \forall x\ y\ a.\ \mathrm{evlis}(\{x\ y\},a) \equiv \{\mathrm{eval}(x,a)\ \mathrm{eval}(y,a)\}\end{aligned}$$
etc.

iv)
$$\begin{aligned}&\vdash \forall x\ y\ a.\ \mathrm{pairlis}(\{x\},\{y\},a) \equiv (x\cdot y)\cdot a\\ &\vdash \forall x1\ x2\ y1\ y2\ a.\ \mathrm{pairlis}(\ \{x1\ x2\},\ \{y1\ y2\},\ a)\\ &\qquad \equiv (x1\cdot y1)\cdot((x2\cdot y2)\cdot a)\end{aligned}$$
etc.

Building on these results, we are able to derive more easily basic lemmas describing the effects of 'eval' and 'apply' on some common constructs. (Again we do not concern ourselves with strictness results but just report on their existence.) We start with three special cases of 'eval' on expressions which do not involve function calls:

v) $\vdash\ \forall x\ a.\ \text{eval}(\text{COND}\circ x,\ a) \equiv \text{evcon}(x,a)$

vi) $\vdash\ \forall x\ a.\ \text{eval}(\{\text{QUOTE } x\},\ a) \equiv \text{islist}(a) \rightarrow x, \bot$

vii) $\text{atom}(X) \equiv \mathbb{T}\ \vdash\ \forall y\ a.\ \text{eval}(X,(X\circ y)\circ a) \equiv \text{islist}(a) \rightarrow y, \bot$

viii) $\text{atom}(X) \equiv \mathbb{T},\ X{=}X1 \equiv \mathbb{F}$

$$\vdash\ \forall y\ y1\ a.\ \text{eval}(X,(X1\circ y1)\circ((X\circ y)\circ a)) \equiv \text{islist}(a) \rightarrow \partial(y1) \rightarrow y, \bot, \bot$$

There is some taste involved in how one states many of these theorems. This last theorem, for instance, could have been written as

$$\text{atom}(X) \equiv \mathbb{T},\ X{=}X1 \equiv \mathbb{F},\ \text{islist}(A) \equiv \mathbb{T},\ \partial(Y1) \equiv \mathbb{T}\ \vdash\ \forall y.\ \text{eval}(X,(X1\circ Y1)\circ((X\circ y)\circ A)) \equiv y$$

The next group of theorems concerns the application of the five standard functions:

ix) $\vdash\ \forall x\ a.\ \text{apply}(\text{CAR},\{x\},a) \equiv \text{islist}(a) \rightarrow \text{hd}(x), \bot$

$\vdash\ \forall x\ a.\ \text{apply}\ (\text{ATOM},\{x\},a) \equiv \text{islist}(a) \rightarrow (\text{atom}(x) \rightarrow \text{T},\text{F}), \bot$

and similar results for CDR, CONS and EQ

x) $\vdash\ \forall x\ a.\ \text{eval}(\{\text{CDR } x\},\ a) \equiv \text{tl}(\text{eval}(x,a))$

$\vdash\ \forall x\ y\ a.\ \text{eval}(\{\text{CONS } x\ y\},\ a) \equiv \text{eval}(x,a)\circ\text{eval}(y,a)$

and similar results for CAR, ATOM and EQ.

Finally there are theorems (or families of theorems) for the cases of 'eval' and 'apply' which involve functions which are given explicitly as LABELed expressions or as LAMBDA expressions:

xi) $\vdash\ \forall n\ f\ x\ a.\ \text{apply}(\{\text{LABEL } n\ f\},x,a) \equiv \text{apply}(f,x,(n\circ f)\circ a)$

xii) ⊢ ∀n f x a. eval({{LABEL n f} x},a) ≡ apply(f,evlis(x,a),(n∘f)∘a)

xiii) ⊢ ∀b a. apply({LAMBDA NIL b},NIL,a) ≡ eval(b,a)
⊢ ∀x y b a. apply({LAMBDA {x} b},{y},a) ≡ eval(b,(x•y)∘a)

etc. for higher arities of the function.

xiv) ⊢ ∀b a. eval({{LAMBDA NIL b}},a) ≡ eval(b,a)
⊢ ∀x y b a. eval({{LAMBDA {x} b} y},a) ≡ eval(b,(x∘eval(y,a))∘a)

CHAPTER 7

Applications of the Theory of Pure LISP

We shall discuss in this brief chapter the application of the semantics of Pure LISP (developed in Chapter 6) to the correctness of several simple LISP functions. The purpose of working these examples is to illustrate some simple techniques that may help in converting LISP functions to the the LCF functionals that they yield via interpretation. The three functions we use need to be defined and discussed anyway because they are used in the LISP interpreter that we discuss in the next chapter. The functions are

i) The NULL function of one argument X; It returns T if X is NIL else returns F; There is no recursion involved.

ii) The EQUAL function of two arguments X,Y; It returns T if X is the same individual as Y; It is recursive but calls no other recursive function internally.

iii) An ASSOC function of two arguments X,A; It returns the first pair in list A whose head is X although if there is no such pair it gives NIL; It is recursive and it makes a call on another recursive function (EQUAL).

Before we discuss the examples in turn, some more axioms must be given (added to the environment axioms of Chapter 4 and the Pure LISP axioms of Chapter 6). We must say that EQUAL, X, A, ASSOC etc. are all names (of functions or parameters) and distinct from each other and from the names LAMBDA, CAR etc. Also it is convenient to have names for the S-expressions which are the bodies of the functions NULL, EQUAL, ASSOC. So:

**AXIOM PL11:

$$\begin{aligned}
&\mathrm{discr}(\mathrm{ASSOC}) > \mathrm{discr}(\mathrm{T}) \equiv \mathbb{T},\\
&\mathrm{discr}(\mathrm{EQUAL}) > \mathrm{discr}(\mathrm{ASSOC}) \equiv \mathbb{T},\\
&\mathrm{discr}(\mathrm{NULL}) > \mathrm{discr}(\mathrm{EQUAL}) \equiv \mathbb{T},\\
&\mathrm{discr}(\mathrm{A}) > \mathrm{discr}(\mathrm{NULL}) \equiv \mathbb{T},\\
&\mathrm{discr}(\mathrm{X}) > \mathrm{discr}(\mathrm{A}) \equiv \mathbb{T},\\
&\mathrm{discr}(\mathrm{Y}) > \mathrm{discr}(\mathrm{X}) \equiv \mathbb{T}
\end{aligned}$$

**AXIOM PL12:

```
Snull ≡ (LAMBDA (X) (COND
            ((ATOM X) (EQ X (QUOTE NIL)))
            ((QUOTE T) (QUOTE F))))
```

**AXIOM PL13:

```
Sequal ≡ (LABEL EQUAL SequalB),
SequalB ≡ (LAMBDA (X Y) (COND
            ((ATOM X) (COND
                ((ATOM Y) (EQ X Y))
                ((QUOTE T) (QUOTE F))))
            ((ATOM Y) (QUOTE F))
            ((EQUAL (CAR X) (CAR Y))
                (EQUAL (CDR X) (CDR Y)))
            ((QUOTE T) (QUOTE F))))
```

```
**AXIOM PL14:
              Sassoc ≡ (LABEL ASSOC SassocB),
              SassocB ≡ (LAMBDA (X A) (COND
                        ((NULL A) (QUOTE NIL))
                        ((EQUAL (CAR (CAR A)) X) (CAR A))
                        ((QUOTE T) (ASSOC X (CDR A)))))
```

7.1. The NULL Function:

The correctness of the NULL function, given by an S-expression above, is succinctly captured in the theorem:

$$\vdash \ \forall e.\ \mathrm{lisp}(\ (\mathrm{Snull}\ e)\ ) \equiv \mathrm{null}(\mathrm{eval}(e)) \rightarrow T, F\ .$$

However, two theorems which are much more useful are:

$$\vdash \ \forall e\ a.\ \mathrm{apply}(\mathrm{Snull},(e),a) \equiv \mathrm{islist}(a) \rightarrow (\mathrm{null}(e) \rightarrow T,F),\bot$$

and

$$\vdash \ \forall e\ a. \mathrm{eval}((\mathrm{Snull}\ e),a) \equiv \mathrm{null}(\mathrm{eval}(e,a)) \rightarrow T,F\ .$$

Actually, these theorems cover only the important and usual case where the function is applied to precisely one argument. A more general result is:

$$\vdash \ \forall x\ a.\ \mathrm{apply}(\mathrm{Snull},x,a) \equiv \mathrm{islist}(a) \rightarrow (\mathrm{null}(\mathrm{hd}(x)) \rightarrow T,F),\bot\ .$$

In fact, all of these theorems are trivial to prove in the LCF system and it suffices to consider just the second of the four. The appropriate attack is with ABSTRaction followed by CASES on 'islist(a)' and '∂(e)'. The only subcase with any interest is the one in which we have '$\mathrm{islist}(a)\equiv\mathbb{T}$, $\partial(e)\equiv\mathbb{T}$'. We use SIMPLification on this and the subgoal we get is

$$\begin{aligned}&\mathrm{atom}(e) \rightarrow (\mathrm{atom}(e)\rightarrow(e=\mathrm{NIL}),\bot)\rightarrow T,F,\\ &\qquad\quad \mathrm{atom}(e) \rightarrow \bot,\, F \;\equiv\; \mathrm{null}(e)\rightarrow T,F\ .\end{aligned}$$

This subgoal happens to be easily provable by CASES on 'atom(e)' but the important thing about it is that it contains no mention of 'eval', 'apply' etc.; It is simply a proposition in LCF involving the equality of two terms which denote individuals and proving this subgoal has nothing whatever to do with the semantics of LISP. The number of substitutions which were ordered by the simplification routine is quite large and so we see we are reaping benefits from having a SIMPSET which was rich in special cases of the LISP primitives.

The NULL function is a good example of the simple (but common) case of a function $\mathbf{F}$ which is just a LAMBDA term and which contains no nested LABEL constructs and uses no unbound variables. As a statement of correctness of $\mathbf{F}$, we will be seeking to prove a theorem that looks like

$$\forall a\ x\ y....\ \mathrm{apply}(\mathbf{F},(x\ y\ ...),a) \equiv \mathrm{islist}(a)\rightarrow G(x,y,...),\bot$$

It is proposed that in proving such a result one attacks with ABSTRaction, does CASES on 'islist(a)' and CASES on the definedness of each of the arguments (x,y,...). If we are lucky all but one of the subgoals are trivial and the nontrivial one SIMPLifies to a subgoal which is quite free of 'eval', 'apply' etc.

7.2. **The EQUAL Function:**

EQUAL is an example of a function which is recursive but does not call any other recursive function internally (i.e. it does not contain any LABEL constructs). Again the statement of correctness is simple and comes in a variety of forms such as:

$$\forall x\ y\ a.\ \mathrm{apply}(\mathrm{Sequal},\{x\ y\},a) \equiv \mathrm{islist}(a) \rightarrow (x=y)\rightarrow T,F,\ \bot\ .$$

Recalling that Sequal and SequalB are the S-expressions that are the whole LISP function and its body respectively, we tackle the above theorem via the lemma:

$$\begin{aligned}\forall x\ y\ a.\ \partial(x)\Rightarrow \partial(y)\Rightarrow\ & \mathrm{assoc}(\mathrm{EQUAL},a)=(\mathrm{EQUAL}\cdot\mathrm{SequalB})\Rightarrow\\ & \mathrm{apply}(\mathrm{SequalB},\ \{x\ y\},\ a) \equiv (x=y)\rightarrow T,F\ \ .\end{aligned}$$

This lemma is appropriately attacked by induction on the structure of either of the arguments of EQUAL since the recursion of this function takes both apart. More specifically we do induction on some occurences of '∂' using an equation that was introduced as an axiom in the Theory of Lists in Chapter 4

$$\partial \equiv [\mu G.[\lambda x.\ \mathrm{atom}(x)\rightarrow \mathbb{T},G(\mathrm{hd}(x))\rightarrow G(\mathrm{tl}(x)),\bot]]$$

The base case is trivial and the other case reduces to a subgoal where we have

$$\begin{aligned}& G(\mathrm{hd}(x))\equiv\mathbb{T},\quad G(\mathrm{tl}(x))\equiv\mathbb{T},\quad \partial(y)\equiv\mathbb{T},\\ & \mathrm{assoc}(\mathrm{EQUAL},a)\equiv(\mathrm{EQUAL}\cdot\mathrm{SequalB}),\\ & \forall x\ y\ a.G(x)\Rightarrow\ \partial(y)\Rightarrow\ \mathrm{assoc}(\mathrm{EQUAL},a)=(\mathrm{EQUAL}\cdot\mathrm{SequalB})\Rightarrow\\ & \qquad\qquad \mathrm{apply}(\mathrm{SequalB},\ \{x\ y\},\ a)\ \equiv\ (x=y)\rightarrow T,F\end{aligned}$$

and we must prove

$$\mathrm{apply}(\mathrm{SequalB}, \{x\ y\}, a) \equiv (x{=}y) \rightarrow T, F\ .$$

The next attack on the problem is by using the definition of SequalB and the SIMPSET which is primed with the nice lemmas that we described in Chapter 6. Simplification does not simplify it to something which is free of 'apply' and 'eval'; the subterms which are the recursive calls on EQUAL are almost intact. However, by doing the CASES arguments that suggest themselves and applying the induction hypothesis we complete the proof of the lemma and then the proof of the main result for EQUAL quickly follows.

The important technique illustrated is that when one has a LISP function F which is an S-expression {LABEL F B} (where B is the body), and we want to establish a theorem that looks like

$$\forall x\ y\ \ldots.\ \mathrm{apply}(F,\{x\ y\ \ldots\},a) \equiv \mathrm{islist}(a) \rightarrow G(x,y,\ldots),\bot$$

then we try to prove a lemma that looks like

$$\begin{aligned}\forall x\ y\ \ldots\ a.\ \partial(x) \Rightarrow \partial(y) \Rightarrow \ldots . \Rightarrow \mathrm{assoc}(F,a) = (F \circ B) \Rightarrow& \\ \mathrm{apply}(B,\{x\ y\ \ldots\},a) \equiv G(x,y,\ldots)&\end{aligned}$$

and we attack the problem using an induction that reflects the computation that function G performs; perhaps we use the definition of G and perhaps we use induction on the structure of an argument of G that it tears apart.

7.3. The ASSOC Function:

We refer back to the start of the chapter for the S-expression form of ASSOC (the S-expression is named Sassoc). We also give here a corresponding Pure LISP function in M-notation

```
assoc[x;a] = [null[a]→NIL;
              equal[car[a];x]→car[x];
              T→assoc[x;cdr[a]]]
```

As shown by its definition, the ASSOC function chosen is recursive and also makes internal use of another recursive function; that is, it has a nested LABEL construct. The correctness results for ASSOC are typified by:

$$\forall x\ y\ a.\ \text{apply}(\text{Sassoc},\langle x\ y\rangle,a) \equiv \text{islist}(a)\rightarrow \text{assoc}(x,y),\ \perp .$$

The recursion aspect is handled in the same manner as it was in the proof of correctness of EQUAL; we prove the lemma

$$\forall x\ y\ a.\ \partial(x)\Rightarrow \partial(y)\Rightarrow \text{assoc}(\text{ASSOC},a)=(\text{ASSOC}\circ\text{SassocB})\Rightarrow \text{apply}(\text{Sassoc},\langle x\ y\rangle,a) \equiv \text{assoc}(x,y)$$

doing it by induction on the second argument of ASSOC. The internal call on the recursive function EQUAL is no problem because we already have the result (last section):

$$\forall x\ y\ a.\ \text{apply}(\text{Sequal},\langle x\ y\rangle,a) \equiv \text{islist}(a)\rightarrow(x=y)\rightarrow T,F,\perp$$

which is great as a simplification rule.

In general when a function contains a call on a recursive function, we prove a correctness result for the sub-function first.

7.4. **Remarks:**

The extraction of meaning functions for LISP functions from their S-expression forms, provided mutual recursion is not involved, seems rather straightforward and the prognosis for automation of the process is good. The simplification mechanism already does a huge amount of the work and it is the author's belief that more effort spent on the scope of the Theory of Pure LISP and further development of the LCF system would make the proofs even easier to generate and comprehend.

Although we have not worked any simple examples of correctness of mutually recursive functions the LISP Interpreter proof in the next chapter involves several case of mutual recursion (and is rather complicated).

CHAPTER 8

The Correctness of an Interpreter

When McCarthy presented an interpreter for Pure LISP he did so in 'm-expression' notation but the report also contained an algorithm for translating m-expressions to S-expressions. Following his prescription (and making the change to 'evcon' recommended in Chapter 6), we present in Figure 8.1 (next two pages) the various functions (that constitute this interpreter) as S-expressions; we also give names to these terms so they are given as an extra axiom (PL15). Note that we still need all the axioms of Chapter 7 (as well as those for Pure LISP and the environment) since EVAL, APPLY etc. make use of NULL, EQUAL and ASSOC.

Note that these functions are oriented towards EVAL being the function called at the top level. In Pure LISP one does not declare the various functions one uses but writes them down in every place they are called except inside of themselves. Hence, as PL15 is written, Sapply must just be considered a subexpression of Seval; 'lisp((Sapply x))' will be undefined for all S-expressions x that require a call of EVAL. Similar remarks hold for Sevlis and Sevcon. If it was desired that APPLY be the main function (as in the 'evalquote' model of the top level) then one could change (in PL15) the 'EVAL's in 'SapplyB' to 'Seval' and the 'Sapply' in 'Seval' to 'APPLY'.

**AXIOM PL15:

```
Seval = (LABEL EVAL SevalB)
SevalB = (LAMBDA (E A) (COND
        ((ATOM E) (CDR (Sassoc E A)))
        ((ATOM  (CAR E)) (COND
           ((EQ (CAR E) (QUOTE QUOTE)) (CAR (CDR E)))
           ((EQ (CAR E) (QUOTE COND)) (Sevcon (CDR E) A))
           ((QUOTE T) (Sapply (CAR E) (Sevlis (CDR E) A) A))))
        ((QUOTE T) (Sapply (CAR E) (Sevlis (CDR E) A) A))))


Sapply = (LABEL APPLY SapplyB)
SapplyB = (LAMBDA (FN X A) (COND
        ((ATOM FN) (COND
           ((EQ FN (QUOTE CAR)) (CAR (CAR X)))
           ((EQ FN (QUOTE CDR)) (CDR (CAR X)))
           ((EQ FN (QUOTE CONS))
                   (CONS (CAR X) (CAR (CDR X))))
           ((EQ FN (QUOTE ATOM)) (ATOM (CAR X)))
           ((EQ FN (QUOTE EQ))
                   (EQ (CAR X) (CAR (CDR X))))
           ((QUOTE T) (APPLY (EVAL FN A) X A))))
        ((EQ (CAR FN) (QUOTE LAMBDA))
                (EVAL (CAR (CDR (CDR FN)))
                         (Spairlis (CAR (CDR FN)) X A)))
        ((EQ (CAR FN) (QUOTE LABEL))
                (APPLY (CAR (CDR (CDR FN)))  X
                         (CONS (CONS (CAR (CDR FN))
                                  (CAR (CDR (CDR FN)))) A)))))
```

Figure 8.1a - S-expression Form of the Interpreter.

Reproduced from best available copy.

```
Sevcon = (LABEL EVCON SevconB)
SevconB = (LAMBDA (C A) (COND
        ((EVAL (CAR (CAR C)) A)
                (EVAL (CAR (CDR (CAR C))) A))
        ((EQ (EVAL (CAR (CAR C)) A) (QUOTE F))
                (EVCON (CDR C) A))))


Sevlis = (LABEL EVLIS SevlisB)
SevlisB = (LAMBDA (M A) (COND
        ((Snull M) (QUOTE NIL))
        ((QUOTE T) (CONS (EVAL (CAR M) A)
                        (EVLIS (CDR M) A))))))


Spairlis = (LABEL PAIRLIS SpairlisB)
SpairlisB = (LAMBDA (X Y A) (COND
          ((Snull X) A)
          ((QUOTE T) (CONS (CONS (CAR X) (CAR Y))
                (PAIRLIS (CDR X) (CDR Y) A)))))
```

Figure 8.1b - S-expression Form of the Interpreter (ctd).

Before we discuss the correctness of these functions we must give yet one more axiom to introduce the various function names and formal parameter names for the functions:

PAIRLIS, APPLY, ASSOC, EVCON, EVLIS, EVAL, FN, C, E, M .

We do this in the same way as we introduced particular names in Chapters 2 and 3; that is:

**AXIOM PL16:

$$\begin{aligned}&\mathrm{discr}(\mathrm{PAIRLIS}) > \mathrm{discr}(\mathrm{Y}) \equiv \mathbb{T},\\ &\mathrm{discr}(\mathrm{APPLY}) > \mathrm{discr}(\mathrm{PAIRLIS}) \equiv \mathbb{T},\\ &\text{etc.}\end{aligned}$$

8.1. Meaning of PAIRLIS:

The PAIRLIS function is similar in structure to the ASSOC function (see previous chapter) in that it is recursive and has an internal call to another function. It is not involved in the mutual recursion that is exhibited by EVAL etc. so we are able to give a meaning function for it just as we did with ASSOC. It should come as no surprise to learn that the function induced by PAIRLIS under interpretation is the 'pairlis' function which is part of the axioms for Pure LISP.

A convenient statement of correctness for the PAIRLIS function is the following:

$$\forall x\ y\ a\ al.\ \mathrm{apply}(\$\mathrm{pairlis}, \langle x\ y\ a\rangle, al)$$
$$\equiv \mathrm{islist}(al) \rightarrow a(y) \rightarrow \mathrm{pairlis}(x,y,a),\ \perp,\ \perp\ .$$

Care should be used to avoid confusion (here and in the rest of the chapter) when two A-lists appear in a theorem; one will be used in the LCF interpretation of the interpreter functions (such as EVAL and EVLIS) and the other is a parameter of these functions. In the case we have here, PAIRLIS needs an A-list as a parameter and 'apply' need an A-list to interpret the Interpreter function PAIRLIS.

8.2. Important Lemmas:

The big problem with EVAL, APPLY, EVCON and EVLIS is that they are mutually recursive; each of APPLY, EVCON and EVLIS call EVAL and EVAL calls the other three. Although it is rather complicated as an example, it is hoped that the proof of correctness of EVAL will give some insight to the rather common phenomenon of mutual recursion.

We now present the main correctness theorem for the S-expression form of the Pure LISP interpreter:

$$\forall e\ a\ al.\ \mathrm{apply}(Seval,\{e\ a\},al) \equiv \mathrm{islist}(al) \rightarrow \mathrm{eval}(e,a),\bot$$

and we will also seek the auxiliary results:

**1 $\mathrm{assoc}(EVAL,al) \equiv (EVAL \cdot SevalB)$

$$\vdash\ \forall e\ a.\ \mathrm{apply}(SevalB,\{e\ a\},al) \equiv \mathrm{eval}(e,a)$$

**2 $\mathrm{assoc}(EVAL,al) \equiv (EVAL \cdot SevalB)$

$$\vdash\ \forall c\ a.\ \mathrm{apply}(Sevcon,\{c\ a\},al) \equiv \mathrm{evcon}(c,a)$$

$$\begin{aligned} **3 \quad & \mathrm{assoc}(EVAL,al) \equiv (EVAL \cdot SevalB) \\ & \qquad \vdash \forall m\ a.\ \mathrm{apply}(Sevlis,\{m\ a\},al) \equiv \mathrm{evlis}(m,a) \end{aligned}$$

$$\begin{aligned} **4 \quad & \mathrm{assoc}(EVAL,al) \equiv (EVAL \cdot SevalB) \\ & \qquad \vdash \forall fn\ x\ a.\ \mathrm{apply}(Sapply,\{fn\ x\ a\},al) \equiv \mathrm{apply}(fn,x,a) \end{aligned}$$

(Note that by a property of 'assoc' we can deduce from '$\partial(\mathrm{assoc}(X,al)) \equiv \mathbb{T}$' the fact '$\mathrm{islist}(al) \equiv \mathbb{T}$').

Without seeking prior motivation, consider just the 'evlis' function (because it is the simplest) and the following proposition:

$$\begin{aligned} **5 \quad & \mathrm{islist}(a) \equiv \mathbb{T}, \\ & \mathrm{assoc}(EVLIS,al) \equiv (EVLIS \cdot SevlisB), \\ & \mathrm{assoc}(EVAL,al) \equiv (EVAL \cdot SevalB), \\ & \quad \vdash \forall m.\ \mathrm{apply}(Sevlis,\{m\ a\},al) \\ & \qquad \equiv [\lambda al1.\ \mathrm{null}(m) \rightarrow NIL, \\ & \qquad\qquad \mathrm{apply}(SevalB,\{\mathrm{hd}(m)\ a\},al1) \cdot \mathrm{apply}(SevlisB,\{\mathrm{tl}(m)\ a\},al1)] \\ & \qquad\qquad\qquad (M \cdot m) \cdot ((A \cdot a) \cdot al) \quad . \end{aligned}$$

One cannot help but notice a strong resemblance between the consequent of this equation and the recursive equation that 'evlis' is the least fixed point of:

$$\mathrm{evlis} \equiv [\lambda m\ a.\ \mathrm{null}(m) \rightarrow NIL, \mathrm{eval}(\mathrm{hd}(m),a) \cdot \mathrm{evlis}(\mathrm{tl}(m),a)\ ] \ .$$

This lemma (**5) is aptly characterised as a statement of 'relative correctness' of EVLIS since if the function EVAL were correct (i.e. obeys result **1) then a simple induction will transform it into the correctness statement **3.

The proof of the lemma (**5) is conceptually very simple involving only the multiple application of the definition of 'apply' (and the other interpreting functions) and is generable interactively very easily: it involves less than 30 steps (mainly CASES and SIMPLifications) although there are hundreds of behind-the-scenes substitutions performed by the simplification algorithm.

Similar lemmas are provable for the other 3 functions (eval, apply and evcon) and we state all four results as Figure 8.2 (next two pages). These theorems should be compared closely with those of Figure 8.3 to note the correspondence of structure. The four proofs are almost mechanical since they involve primarily obvious CASES arguments and SIMPLifications. The proof of the lemma involving APPLY is the longest being about 60 steps.

8.3. Informal Proof of Interpreter Correctness:

Now, speaking quite informally, and omitting any discussion of the definedness (or listness) of arguments of EVAL, APPLY etc. it is readily seen that these four lemmas can serve as a basis for **computing** values of the function

[λe a al. apply(Seval,(e a),al)]

just as the equations of Fig 8.3 can serve as a basis for computing values of 'eval'. For example,

eval((ATOM (QUOTE X)), NIL)
computes through
apply(ATOM,evlis(((QUOTE X)),NIL),NIL)

```
islist(a) ≡ 𝕋 ⊢
    ∀m al . assoc(EVLIS,al)=(EVLIS·SevlisB)⇒
                 assoc(EVAL,al)=(EVAL·SevalB) ⇒
       apply(SevlisB,m·(a·NIL),al) ≡ null(m)→NIL,
           (apply(SevalB,hd(m)·(a·NIL),(M·m)·((A·a)·al))
               ·apply(SevlisB,tl(m)·(a·NIL),(M·m)·((A·a)·al)))


islist(a) ≡ 𝕋 ⊢
   ∀fn x al . assoc(APPLY,al)=(APPLY·SapplyB) ⇒
                 assoc(EVAL,al)=(EVAL·SevalB) ⇒
    apply(SapplyB,fn·(x·(a·NIL)),al) ≡ (fn=CAR)→hd(hd(x)),
      ((fn=CDR)→tl(hd(x)),
      ((fn=CONS)→(hd(x)·hd(tl(x))),
      ((fn=ATOM)→(atom(hd(x))→T,F),
      ((fn=EQ)→([λx y .atom(x)→(atom(y)→(x=y),⊥),⊥]
                                      (hd(x),hd(tl(x)))→T,F),
      (atom(fn)→apply(SapplyB,apply(SevalB,fn·(a·NIL),
                     (FN·fn)·((X·x)·((A·a)·al)))·(x·(a·NIL)),
                                      (FN·fn)·((X·x)·((A·a)·al))),
      ((hd(fn)=LAMBDA)→apply(SevalB,
                     hd(tl(tl(fn)))·(pairlis(hd(tl(fn)),x,a)·NIL),
                                    (FN·fn)·((X·x)·((A·a)·al))),
      ((hd(fn)=LABEL)→apply(SapplyB,
             hd(tl(tl(fn)))·(x·(((hd(tl(fn))·hd(tl(tl(fn))))·a)·NIL)),
                             (FN·fn)·((X·x)·((A·a)·al))),⊥)))))))
```

Figure 8.2a - Some Lemmas about SevlisB and SapplyB.

```
islist(a) ≡ 𝕋  ⊢
    ∀c al . assoc(EVCON,al)=(EVCON·SevconB) ⇒
            assoc(EVAL,al)=(EVAL·SevalB) ⇒
    apply(SevconB,c·(a·NIL),al) ≡
        (apply(SevalB,hd(hd(c))·(a·NIL),(C·c)·((A·a)·al))=T)→
                apply(SevalB,hd(tl(hd(c)))·(a·NIL),(C·c)·((A·a)·al)),
        ((apply(SevalB,hd(hd(c))·(a·NIL),(C·c)·((A·a)·al))=F)→
                apply(SevconB,tl(c)·(a·NIL),(C·c)·((A·a)·al)),⊥)


islist(a) ≡ 𝕋  ⊢
    ∀x al. assoc(EVAL,al)=(EVAL·SevalB) ⇒
    apply(SevalB,x·(a·NIL),al) ≡ (atom(x)→tl(assoc(x,a)),
            ((hd(x)=QUOTE)→hd(tl(x)),
            ((hd(x)=COND)→apply(SevconB,tl(x)·(a·NIL),
                            (EVCON·SevconB)·((E·x)·((A·a)·al))),
            apply(SapplyB,hd(x)·(apply(SevlisB,tl(x)·(a·NIL),
                        (EVLIS·SevlisB)·((E·x)·((A·a)·al)))·(a·NIL)),
                (APPLY·SapplyB)·((E·x)·((A·a)·al)))))),⊥
```

Figure 8.2b - Some Lemmas about SevconB and SevalB.

```
⊢ eval ≡ [λe a.
        islist(a)→
           (atom(e) → tl(assoc(e,a)),
           hd(e)=QUOTE → hd(tl(e)),
           hd(e)=COND → evcon(tl(e),a),
           apply(hd(e), evlis(tl(e),a), a)), ⊥]

⊢ evcon ≡ [λc a. (eval(hd(hd(c)),a)=T) → eval(hd(tl(hd(c))),a),
              (eval(hd(hd(c)),a)=F) → evcon(tl(c),a) , ⊥]

⊢ apply ≡ [λfn x a. ∂(x)→ islist(a)→
               (fn=CAR) → hd(hd(x)),
               (fn=CDR) → tl(hd(x)),
              (fn=CONS) → hd(x)∘hd(tl(x)),
              (fn=ATOM) → atom(hd(x)) → T, F,
                (fn=EQ) → [λx y. atom(x)→atom(y)→(x=y), ⊥, ⊥]
                                     (hd(x),hd(tl(x)))→ T, F,
                atom(fn) → apply(eval(fn,a),x,a),
          (hd(fn)=LAMBDA) → eval(hd(tl(tl(fn))),
                                pairlis(hd(tl(fn)),x,a)),
           (hd(fn)=LABEL) → apply(hd(tl(tl(fn))), x,
                               ((hd(tl(fn))∘hd(tl(tl(fn))))∘a)),
                         ⊥, ⊥, ⊥ ]

⊢ evlis ≡ [λm a. null(m)→NIL, eval(hd(m),a) ∘ evlis(tl(m),a)]
```

Figure 8.3 - Some Lemmas about eval, apply, evlis & evcon.

```
and
                 apply(ATOM,⟨eval(⟨QUOTE X⟩,NIL)⟩,NIL)
and
                 apply(ATOM,⟨X⟩,NIL)
to
                 T .
```

Similarly,

```
                 apply(SevalB,⟨⟨ATOM ⟨QUOTE X⟩⟩ NIL⟩, NIL)
computes through
                 apply(SapplyB,⟨ATOM
                        apply(SevlisB,⟨⟨⟨QUOTE X⟩⟩ NIL⟩,al1) NIL⟩,al1)
and
                 apply(SapplyB,⟨ATOM
                        ⟨apply(SevalB,⟨⟨QUOTE X⟩ NIL⟩,al2)⟩ NIL⟩,al1)
and
                 apply(SapplyB,⟨ATOM ⟨X⟩ NIL⟩,AL1)
to
                 T .
```

In all such examples, the computation terminates when there is no applicable lemma; this will be just when there is no more instances of the interpreting functions ('apply' etc.) and if the computation does not terminate then the result will be ⊥.

It should be apparent that if we do the computations for 'eval(e,a)' and 'apply(SevalB,⟨e a⟩,al)' then because of the structural similarity between the two sets of computation rules, those two computations will proceed in parallel just as they did in the above example. Moreover, if one of these computations terminates with a certain result then so will the other and if one never halts then neither will the other. That completes the informal proof.

8.4. Interpreter Correctness in LCF:

The above informal proof suggests an attack on the desired main results (**1 to **4) using the results of Figure 8.2 and computation induction. It is appropriate to do induction on the definition of 'eval' but we notice that in terms of recursion on the computation of 'eval' (and 'apply' etc.) the left hand sides of the desired results compute much slower than the right hand sides. This is because the interpretation of each expression is done directly on the right hand side but indirectly (via interpretation of EVAL, APPLY, EVLIS or EVCON) on the left. Thus in doing the proof we are forced to break each of the four equivalences into two 'inequivalences':

assoc(EVAL,AL)≡(EVAL·SevalB)
⊢ ∀x a. apply(SevalB, (x a), AL) ⊑ eval(x,a) ,

assoc(EVAL,AL)≡(EVAL·SevalB)
⊢ ∀x a. eval(x,a) ⊑ apply(SevalB, (x a), AL) ,

assoc(EVAL,AL)≡(EVAL·SevalB)
⊢ ∀c a. apply(Sevcon, (c a), AL) ⊑ evcon(c,a)

etc.

8.5. **Partial Correctness:**

We first report on the proof of the inequality

**6 $$\vdash \forall x\ a\ al.\ \mathrm{assoc}(EVAL,al)=(EVAL\cdot SevalB)\Rightarrow \\ \mathrm{eval}(x,a) \sqsubseteq \mathrm{apply}(SevalB, \langle x\ a\rangle, al)$$

or, using the predicate $Q\equiv[\lambda al.\mathrm{assoc}(EVAL,al)=(EVAL\cdot SevalB)]$,

$$\vdash \forall x\ a\ al.\ Q(al)\Rightarrow \mathrm{eval}(x,a) \sqsubseteq \mathrm{apply}(SevalB,\langle x\ a\rangle,al)\ .$$

We consider this proof in greater detail than any previous one because it is quite complex involving several nested inductions. The outermost induction uses the definition of 'eval' and the inner ones correspond to the definitions of 'apply', 'evlis' and 'evcon'.

First we rewrite the definition of 'eval' from Fig 6.1 in the form 'eval $\equiv [\mu B.P(B)]$' thus defining functional 'P' which is free of B. Then we attack the goal with an induction that uses this equation, to give the subgoals:

i) $\forall x\ a\ al.\ Q(al)\Rightarrow \bot(x,a) \sqsubseteq \mathrm{apply}(SevalB,\langle x\ a\rangle,al)$,

ii) $$\forall x\ a\ al.\ Q(al)\Rightarrow B(x,a) \sqsubseteq \mathrm{apply}(SevalB,\langle x\ a\rangle,al) \\ \vdash \forall x\ a\ al.\ Q(al)\Rightarrow P(B,x,a) \sqsubseteq \mathrm{apply}(SevalB,\langle x\ a\rangle,al)\ .$$

Now subgoal (i) is trivial by SIMPLification. We attack the second by ASSUMing the antecedent, doing PREFix removal in the consequent, CASES on 'atom(x)' and SIMPLification. We therefore have an induction hypothesis assumed and one (complex) subgoal corresponding to the interesting case where

$$\begin{aligned}&\text{' islist(a)} \equiv \mathbb{T},\\ &\quad \text{assoc(EVAL,al)} \equiv \text{(EVAL}\cdot\text{SevalB)},\\ &\qquad \text{atom(x)} \equiv \mathbb{F}\ \text{'}\ .\end{aligned}$$

We further attack this subgoal by CASES on 'hd(x)=QUOTE' and CASES on 'hd(x)≡COND' and by using some monotonicity theorems we get four subgoals which are shown in Figure 8.4 (next page).

The key to proving each of these is an initial induction; in the last we use the structure of the first argument of EVLIS; in the first three we do induction on the fixed point term that appears on the left hand side. Each proof then proceeds by CASES, SIMPLification and USEs of monotonicity theorems (extensive use is also made of the lemmas of Fig 8.2).

Having established **6, it is easy to prove the complementary results.

$$\vdash\ \forall x\ a\ al.\ Q(al) \Rightarrow\ \text{evlis}(x,a) \sqsubseteq \text{apply}(\text{SevlisB},\{x\ a\},al)\ ,$$

$$\vdash\ \forall x\ a\ al.\ Q(al) \Rightarrow\ \text{evcon}(x,a) \sqsubseteq \text{apply}(\text{SevconB},\{x\ a\},al)\ ,$$

$$\vdash\ \forall x\ a\ al.\ Q(al) \Rightarrow\ \text{apply}(x,a) \sqsubseteq \text{apply}(\text{SapplyB},\{x\ a\},al)$$

and also

$$**7 \quad \text{islist(AL)} \equiv \mathbb{T},\ \partial(\text{eval}(X,A)) \equiv \mathbb{T}\ \vdash\ \text{eval}(x,a) \equiv \text{apply}(\text{Seval},\{x,a\},AL)$$

which is a statement of **Partial Correctness** for EVAL, since it says that for any expression which can be evaluated in the context of a certain association list

```
GOAL ∀X A AL.  islist(A)⇒ assoc(EVAL,AL)=(EVAL·SevalB)⇒
        ∂(X)⇒ assoc(EVCON,AL)=(EVCON·SevconB)⇒
            [μG.[λc a. (B(hd(hd(c)),a)=T)→B(hd(tl(hd(c))),a),
                 (B(hd(hd(c)),a)=F)→G(tl(c),a),⊥]](X,A)
                                ≡ apply(SevconB,X·(A·NIL),AL);

GOAL ∀X A AL.  islist(A)⇒ assoc(EVAL,AL)=(EVAL·SevalB)⇒
        ∂(X)⇒ assoc(EVLIS,AL)=(EVLIS·SevlisB)⇒
            [μG.[λm a. null(m)→NIL,B(hd(m),a)·G(tl(m),a)]](X,A)
                                ≡ apply(SevlisB,X·(A·NIL),AL);

GOAL ∀FN X A AL.  islist(A)⇒ assoc(EVAL,AL)=(EVAL·SevalB)⇒
        ∂(X)⇒ assoc(APPLY,AL)=(APPLY·SapplyB)⇒
            [μG.[λfn x a. ∂(x) → islist(a) →
                      (fn=CAR) → hd(hd(x)),
                      (fn=CDR) → tl(hd(x)),
                    (fn=CONS) → hd(x)·hd(tl(x)),
                   (fn=ATOM) → atom(hd(x)) → T, F,
                        (fn=EQ) → [λx y. atom(x)→atom(y)→(x=y), ⊥, ⊥]
                                           (hd(x),hd(tl(x)))→ T, F,
                         atom(fn) → G(B(fn,a),x,a),
         (hd(fn)=LAMBDA) → B(hd(tl(tl(fn))), pairlis(hd(tlfn)),x,a)),
           (hd(fn)=LABEL) → G(hd(tl(tl(fn))),x,
                                  ((hd(tl(fn))·hd(tl(tlfn))))·a)),
                                    ⊥, ⊥, ⊥ ]](FN,X,A)

                                ≡ apply(SapplyB,FN·(X·(A·NIL)),AL);

GOAL ∀X A AL.  islist(A)⇒  assoc(EVAL,AL)=(EVAL·SevalB)⇒
        ∂(X)⇒ assoc(EVLIS,AL)=(EVLIS·SevlisB)⇒
                    islist(apply(SevlisB,X·(A·NIL),AL)) ≡ 𝕋;
```

Figure 8.4 - The Important Partial Correctness Subgoals.

(of variable bindings), the function induced by Seval (under interpretation) will correctly evaluate it.

It remains only to comment that the total amount of proof generated so far in this proof of correctness of the interpreter is quite large and has pushed the LCF system to its limits. The proofs of the lemmas of Fig. 8.2 each required a seperate core image and the proof mentioned in this section required the largest core image possible (128K of which 50K is the LCF system). The main reason for the gross size of the proofs was the magnitude of the formulae involved but there were over a thousand steps involved too. Moreover the CPU time involved was rather large (just over a hundred minutes) reflecting a huge amount of work done by simplification - many thousands of substitutions automatically performed. It must be stressed that were it not for the partial automation afforded by the simplification mechanism of LCF, such a formal proof would not have been possible.

8.6. Total Correctness:

We know from our informal reasoning that the '$\partial(\mathrm{eval}(X,A))\equiv\mathbb{T}$' condition of (**7) can be dropped to give

$$\mathrm{islist}(AL)\equiv\mathbb{T}\ \vdash\ \forall x\ a.\mathrm{eval}(x,a)\equiv\mathrm{apply}(\mathrm{Seval},\{x\ a\},AL)$$

but to establish this formally we need yet to prove the other half of (**1), namely:

$$\forall x\ a\ al.\ \ \mathrm{assoc}(\mathrm{EVAL},al)\equiv(\mathrm{EVAL}\cdot\mathrm{SevalB})\Rightarrow \\ \mathrm{apply}(\mathrm{SevalB},\{x\ a\},al)\sqsubseteq\mathrm{eval}(x,a)\ .$$

This goal is naturally tackled by first expanding the left hand side a little so that the 'apply' vanishes and we have an 'eval' there Instead. Remembering that SevalB is a LAMBDA term we actually get the subgoal

$$\begin{aligned}&\forall x\ a\ al.\ \ \mathrm{assoc}(\mathrm{EVAL},al)=(\mathrm{EVAL}\cdot \mathrm{SevalB})\Rightarrow\\ &\qquad \mathrm{eval}(\mathrm{hd}(\mathrm{tl}(\mathrm{tl}(\mathrm{SevalB}))),(\mathrm{X}\cdot x)\cdot((\mathrm{A}\cdot a)\cdot al)) \sqsubseteq \mathrm{eval}(x,a)\end{aligned}$$

which is appropriately attacked by induction on the definition of 'eval'.

Once the induction is initiated, we are then faced with the work of breaking down the structure of the S-expressions SevalB, SapplyB etc. that appear In the left hand side before we can hope to apply the inductive hypothesis. However, in the subgoal to be proved there is NO occurrences of 'eval', 'apply' etc. There is thus little chance to use SIMPLification since

(a) the theorems we have found so useful (so far) have 'apply', 'eval' etc. on the left hand side.

(b) we are forced to deal with inequalities since the results we must use to break down the left hand side are lemmas such as

$$\text{'}\ B \sqsubseteq \mathrm{eval}\ \vdash\ \forall x\ a.\ B(\{\mathrm{CAR}\ x\},a) \sqsubseteq \mathrm{hd}(B(x,a))\ \text{'}$$

instead of using theorems of the form

$$\text{'}\vdash\ \forall x\ a.\ \mathrm{eval}(\{\mathrm{CAR}\ x\},a) \equiv \mathrm{hd}(\mathrm{eval}(x,a))\ \text{'}\ .$$

At this point, it appears that to pursue the current objective will demand repeating all the work which preceded the proof of the first half of (**1) in a slightly more general form (as illustrated by the last 2 theorems). Furthermore to complete the proof we are faced with an approximately parallel proof (to that described in the last section) but where SIMPLification was used before we will need to use monotonicity results. Now since the current LCF system is so biased towards equalities, the second half of the proof would be extremely tedious using the present system.

Because of this argument, the formal proof of the total correctness of the EVAL function was not carried out. It can again be given consideration when a version of the LCF system is available which can give as much assistance with monotonicity arguments as the current system gives with substitutions.

CHAPTER 9

Compiler Correctness (I) - Language Definitions

In this chapter, we describe axiomatically based theories of the source and target languages of the simple compiler LCom0. We cannot apply the Theory of Pure LISP (except by way of example); instead we must build an alternative (albeit similar) set of axioms and theorems for LCom0 LISP (so-called). For each of the languages, the formal definition will be preceded by an informal description.

9.1. **Extensions to the Environment:**

As in Chapter 6, we precede axiomatisation of languages with some extensions to the environment described in Chapter 4. We identify the axioms introduced in this section by names of the form EEn.

**AXIOM EE1:

$$\begin{aligned} &\forall x.\ \mathrm{isname}(x)\rightarrow \mathrm{isint}(x)\rightarrow\bot,\ \mathrm{atom}(x)\rightarrow x,\bot,\\ &\qquad\qquad \mathrm{isint}(x)\rightarrow \mathrm{atom}(x)\rightarrow x,\bot,\ x \ \equiv x \end{aligned}$$

$$\forall x.\ \partial(\mathrm{discr}(x))\Rightarrow\ \mathrm{isname}(x) \equiv \mathbb{T}$$

$$\forall x.\ \mathrm{isname}(x)\Rightarrow\ \mathrm{discr}(\mathrm{gensym}(x))>\mathrm{discr}(x) \equiv \mathbb{T}$$

$$\begin{aligned} \mathrm{isSexprn} \equiv [\mu G.[\lambda x.\ &\mathrm{null}(x)\rightarrow\mathbb{T},\ \mathrm{isname}(x)\rightarrow\mathbb{T},\\ &\mathrm{isint}(x)\rightarrow\mathbb{T},\ \mathrm{atom}(x)\rightarrow\mathbb{F},\\ &\quad G(\mathrm{hd}(x))\rightarrow G(\mathrm{tl}(x)),\mathbb{F}]] \end{aligned}$$

These axioms introduce names (in general), define S-expressions and provide the appropriate properties of the functions 'discr' and 'gensym'. 'discr' is the same function that was described in Chapter 6 but is specified more completely by these axioms. The important property of 'gensym' is that it maps names onto names in such a way that if we build sequences of names by successive application of the function then no item appears more than once.

We also have many specific names to introduce and the technique for doing this has been illustrated several times so we will just point out the effects of several axioms:

**AXIOM EE2 Introduces the reserved words of the source language of the compiler (a subset of LISP):
LAMBDA, QUOTE, COND, AND, OR, T .

**AXIOM EE3 Introduces the built-in function of the LISP subset:
GREATERP, NUMBERP, GENSYM, EQUAL, MINUS, TIMES, ATOM, CONS, PLUS, CAR. CDR, NOT .

**AXIOM EE4 Introduces the names of some basic LISP functions:
DIFFERENCE, APPEND, LENGTH, ISLIST, APPN2, ASSOC, LESSP, LIST, NULL .

**AXIOM EE5 Introduces the special words of the target language:
JUMPE, JUMPN, MOVEI, CALL, JRST, MOVE, POPJ, PUSH, SUB, C, E, P .

**AXIOM EE6 Introduces names of the compiler functions:
COMPANDOR, COMBOOL, COMCOND, COMPEXP, COMPLIS, LOADAC, MKPUSH, COMP, PRUP .

**AXIOM EE7 Introduces names that are used as formal parameters:
VARS, EXP, FLG, VPR, FN, L1, L2, NL, K, L, M, N, U, X, Y, Z .

9.2. LCom0 LISP:

It was mentioned that LCom0 (McCarthy's compiler discussed by London in [13]) compiles a certain subset of LISP which we will call LCom0 LISP. It should be noted that LCom0 is also **written** in LCom0 LISP.

9.2.1. Informal Description

The language (LCom0 LISP) is rather similar in scope to Pure LISP but the few differences are rather important; in LCom0 LISP:

i) the AND and OR constructions of LISP 1.5 are available;

ii) falsehood is represented as NIL (as opposed to F in Pure LISP) and (although most predicates will return either 'T' or NIL) tests for truth are tests of inequality with NIL;

iii) NIL evaluates to NIL;

iv) there is no LABEL construction and no functional arguments;

v) functions are introduced by fiat at the top level (and there will be a global A-list for function definitions);

vi) S-expressions are based on integers as well as NIL and names;

vii) the built-in functions are CAR, CDR, CONS, ATOM, EQUAL, LIST, NOT, PLUS TIMES, MINUS, NUMBERP, GREATERP and GENSYM; (These functions are the same as in regular LISP except that 'GENSYM' takes a name as input rather than remembering the last name it generated.)

9.2.2. Formal Description

Figure 9.1 gives an axiom (SL1) in which the main functions of an interpretive semantics for LCom0 LISP are defined and Figure 9.2 completes the axiomatisation of LCom0 LISP (with axiom SL2) by giving the meanings of the built-in functions (CAR, CDR etc.). (We identify the axioms related to the LISP subset by names of the form 'SLn' where 'SL' denotes 'Source Language'.) This formal description of the language parallels the definition of Pure LISP semantics so we will avoid lengthy discussion. However, we will emphasize that there are two A-list parameters for 'eval' etc.; the first is used to store variable bindings and the second (constant through the levels of recursion) gives function definitions. If the equations (of Figure 9.1) are a little hard to follow then a glance at Figure 9.3 might help since it shows the recursive equations of which 'eval', 'apply' etc. are the mutually least fixed points.

```
**AXIOM SL1:
    eval ≡ [μB. evalF(B)],
    evalF ≡ [λB x vb fl. ∂(vb) → ∂(fl) →
                  null(x) → NIL,
                  isint(x) → x,
                  isname(x) → tl(assoc(x,vb)),
                  atom(x) → ⊥,
                  hd(x)=QUOTE → hd(tl(x)),
                  hd(x)=COND  → [μG.evconF(B,G)](tl(x),vb,fl),
                  hd(x)=AND   → [μG.evandF(B,G)] (tl(x),vb,fl),
                  hd(x)=OR    → [μG.evorF(B,G)] (tl(x),vb,fl),
                  [μG.applyF(B,G)]
                       ( hd(x), [μG.evlisF(B,G)](tl(x),vb,fl),vb,fl),
                                                         ⊥, ⊥],
    evcon ≡ [μG. evconF(eval,G)],
    evconF ≡ [λF G x vb fl. null(F(hd(hd(x)),vb,fl))→G(tl(x),vb,fl),
                                                 F(hd(tl(hd(x))),vb,fl)],
    evand ≡ [μG. evandF(eval,G)],
    evandF ≡ [λF G x vb fl. null(x)→T, null(F(hd(x),vb,fl))→NIL,G(tl(x),vb,fl)],

    evor  ≡ [μG. evorF(eval,G)],
    evorF ≡ [λF G x vb fl. null(x)→NIL, null(F(hd(x),vb,fl))→G(tl(x),vb,fl),T],

    apply ≡ [μG. applyF(eval,G)],
    applyF ≡ [λF G fn x vb fl. ∂(x) → ∂(vb) → ∂(fl) →
                isBF(fn) → applyBF(fn,x),
                isname(fn) → G(tl(assoc(fn,fl)),x,NIL,fl),
                (hd(fn)=LAMBDA)→ F(hd(tl(tl(fn))),pairlis(hd(tl(fn)),x,vb),fl),
                                                    ⊥, ⊥, ⊥, ⊥ ],
    evlis ≡ [μG. evlisF(eval,G)],
    evlisF ≡ [λF G m vb fl. null(m)→NIL, F(hd(m),vb,fl)·G(tl(m),vb,fl)],

    pairlis ≡ [μG.[λx y vb. null(x) → vb, (hd(x)·hd(y))·G(tl(x),tl(y),vb)]]
```

Figure 9.1 - Axioms for LCom0 LISP.

```
**AXIOM SL2:
      isBF ≡ [λx. (x=CAR)→𝕋,   (x=CONS)→𝕋,    (x=MINUS)→𝕋,
                  (x=CDR)→𝕋,   (x=PLUS)→𝕋,    (x=GENSYM)→𝕋,
                  (x=NOT)→𝕋,   (x=EQUAL)→𝕋,   (x=NUMBERP)→𝕋,
                  (x=ATOM)→𝕋,  (x=TIMES)→𝕋,   (x=GREATERP)→𝕋,
                  (x=LIST)] ,

      applyBF(CAR) ≡ [λx. hd(hd(x))],

      applyBF(CDR) ≡ [λx. tl(hd(x))],

      applyBF(NOT) ≡ [λx. null(hd(x))→T,NIL],

      applyBF(ATOM) ≡ [λx. atom(hd(x))→T,NIL],

      applyBF(CONS) ≡ [λx. hd(x).hd(tl(x))],

      applyBF(LIST) ≡ [λx. x],

      applyBF(PLUS) ≡ [λx. hd(x)+hd(tl(x))],

      applyBF(EQUAL) ≡ [λx. hd(x)=hd(tl(x))→T,NIL],

      applyBF(TIMES) ≡ [λx. hd(x)*hd(tl(x))],

      applyBF(MINUS) ≡ [λx. mns(hd(x))],

      applyBF(GENSYM) ≡ [λx. gensym(hd(x))],

      applyBF(NUMBERP) ≡ [λx. isint(hd(x))→T,NIL],

      applyBF(GREATERP) ≡ [λx. (hd(x)>hd(tl(x)))→T,NIL]
```

Figure 9.2 - The Built-in Functions of LCom0 LISP.

```
eval ≡ [λ x vb fl. ∂(vb) → ∂(fl) →
               null(x) → NIL,
               isint(x) → x,
               isname(x) → tl(assoc(x,vb)),
               atom(x) → ⊥,
               hd(x)=QUOTE → hd(tl(x)),
               hd(x)=COND  → evconF(tl(x),vb,fl),
               hd(x)=AND   → evandF(tl(x),vb,fl),
               hd(x)=OR    → evorF(tl(x),vb,fl),
               applyF( hd(x), evlisF(tl(x),vb,fl), vb, fl), ⊥, ⊥],

evcon ≡ [λx vb fl. null(eval(hd(hd(x)),vb,fl))→evcon(tl(x),vb,fl),
                                        eval(hd(tl(hd(x))),vb,fl)],

evand ≡ [λx vb fl. null(x)→T, null(eval(hd(x),vb,fl))→NIL,
                                             evand(tl(x),vb,fl)],

evor  ≡  [λx vb fl. null(x)→NIL,
                    null(eval(hd(x),vb,fl))→evor(tl(x),vb,fl),T],

apply ≡ [λfn x vb fl. ∂(x) → ∂(vb) → ∂(fl) →
              isBF(fn) → applyBF(fn,x),
              isname(fn) → apply(tl(assoc(fn,fl)),x,NIL,fl),
              (hd(fn)=LAMBDA)→ eval(hd(tl(tl(fn))),pairlis(hd(tl(fn)),x,vb),fl),
                                                  ⊥, ⊥, ⊥, ⊥ ],

evlis ≡ [λm vb fl. null(m)→NIL, eval(hd(m),vb,fl)·evlis(tl(m),vb,fl)],
```

Figure 9.3 - Relationships Between 'eval', 'apply' etc.

9.2.3. Theory of LCom0 LISP

As we did with Pure LISP, we prepare for applications by developing a 'theory' based on the axioms. We do two things in this regard. First, we define some basic LISP functions (actually the ones we need for the compiler proof) such as DIFFERENCE and LENGTH. Next we assemble a collection of theorems (mainly oriented towards SIMPSET inclusion); we exhibit these as an Appendix.

The definitions of the basic LISP functions that we want are given in Figure 9.4 and are given as the actual entries of the function definition A-list (namely: function-name/function-body pairs).

9.2.4. 'BFD' - Basic Functions Defined

We will never actually construct a function list but we require a predicate which says that all the basic functions are declared in some given function list. ('BFD' is mnemonic for 'Basic Function Defined'):

```
**AXIOM SL3:
        BFD ≡ [λfl. tl(assoc(NULL,fl))=Snull →
                     tl(assoc(DIFFERENCE,fl))=Sdifference →
                      tl(assoc(ISLIST,fl))=Sislist →
                       tl(assoc(ASSOC,fl))=Sassoc →
                        tl(assoc(LENGTH,fl))=Slength →
                         tl(assoc(APPEND,fl))=Sappend,𝔽,𝔽,𝔽,𝔽,𝔽]
```

```
**AXIOM SL4:
     Snull ≡  (LAMBDA (X) (EQUAL X NIL)),

     Sdifference ≡ (LAMBDA (X Y) (PLUS X (MINUS Y))),

     Sislist ≡  (LAMBDA (X) (COND
                          ((NULL X) (QUOTE T))
                          ((ATOM X) NIL)
                          ((QUOTE T) (ISLIST (CDR X))))),

     Sassoc ≡  (LAMBDA (X Y) (COND
                          ((NULL Y) NIL)
                          ((EQUAL X (CAR (CAR Y))) (COND
                                ((ISLIST Y) (CAR Y))))
                          ((QUOTE T) (ASSOC X (CDR Y))))),

     Slength ≡  (LAMBDA (X) (COND
                          ((NULL X) 0)
                          ((QUOTE T) (PLUS 1 (LENGTH (CDR X)))))),

     Sappend ≡  (LAMBDA (X Y) (COND
                          ((NULL X) Y)
                          ((QUOTE T) (CONS (CAR X)
                                            (APPEND (CDR X) Y))))),
```

Figure 9.4 - Some Basic LISP Functions.

9.2.5. Well-Formedness Predicate

We give as Figure 9.5 the definition of a predicate 'iswfe' (IS Well Formed Expression) which tells whether an S-expression is structurally good LISP code. It is important because it will be seen later that LCom0 will be total on inputs that satisfy 'iswfe'. Note that one of the things checked is that functions are not called with more than a certain number **mna** of arguments. Note also that all variables referred to inside a well formed expression must be bound by occurring in the formal parameter list of a LAMBDA term.

9.3. LCom0 LAP - Informal Description:

McCarthy's compiler translates the subset of LISP that we call LCom0 LISP into LAP - a special version of PDP10 assembly code which is oriented toward LISP compilation (LAP is an acronym for LISP Assembly Program). Of course, only a subset of the PDP10 instruction set is generable by the compiler and so we will be concerned **only** with certain variants of **nine** instructions (given below) although our formal description will allow later and/or more complete specification of the language. Apart from simply considering a subset of LAP we make some simplifying assumptions about the behaviour of the PDP10; we point out these idealisations below.

```
**AXIOM SL4:

    iswfe ≡ [λe. iswf1(e,NIL)],

    iswfle ≡ [λe. iswf4(iswfe,e,NIL)],

    iswf1 ≡ [μG.[λe vl.
                  null(e)→𝕋,
                   (e=T)→𝕋,
                    isint(e)→𝕋,
                     atom(e)→mem(e,vl),
                      (hd(e)=QUOTE)→IsSexprn(tl(e)),
                       (hd(e)=AND)→iswf2(G,tl(e),vl),
                        (hd(e)=OR)→iswf2(G,tl(e),vl),
                         (hd(e)=COND)→iswf3(G,tl(e),vl),
                          atom(hd(e))→ (length(tl(e))>mna)→𝔽,
                                          iswf2(G,tl(e),vl),
                           iswf4(G,e,vl)]],

    iswf2 ≡ [μH.[λG x vl. null(x)→𝕋,
                  G(hd(x),vl)→H(G,tl(x),vl),𝔽]],

    iswf3 ≡ [μH.[λG x vl. null(x)→𝕋,
                G(hd(hd(x)),vl)→ G(hd(tl(x)),vl)→ H(G,tl(x),vl),𝔽,𝔽]],

    iswf4 ≡ [λG x vl. (hd(x)=LAMBDA)→ iswf5(hd(tl(x)))→
                  length(hd(tl(x)))>mna → G(hd(tl(tl(x))),vl&hd(tl(x))),𝔽,𝔽]],

    iswf5 ≡ [μH.[λx. null(x)→𝕋, isname(hd(x))→H(tl(x)),𝔽]]
```

Figure 9.5 - Well-Formedness of LISP expressions.

In our simplified view of the architecture of the PDP10, we take it to be simply a Central Processing Unit and a Memory. The CPU executes lists of instructions and each instruction executed can affect the flow of control in certain ways and/or affect the state of the memory. The memory is an infinite array of words such that every word has an address which is a positive integer. Also the first sixteen words can be used as accumulators or index registers.

We do not want to become involved in the processes of **assembling** or **loading** of LAP. Also, we do not admit the possibility that LAP instructions will be overwritten during the execution of a program. Hence we make the further assumption that LAP code is interpreted directly (symbolically) and not resident in memory in any way.

The contents of words of the PDP10 are usually treated as integers but we also want to represent S-expressions in memory. We do not want to get involved in questions of representation so we just say that there exists a coding of S-expressions into integers. The only thing we specify about the coding is that it is one-to-one and that the coding of NIL is 0. This assumption enables us to avoid any questions related to free-storage management. Further note that there is no bound to the integers that words may contain. Moreover, we assume that the contents of any word is only defined if a value has been written in already.

Now just as an LCom0 LISP program is a collection of LISP functions, we take a LAP program to be a collection of LAP functions; we define a LAP function to be triple (FN,NA,FB) where 'FN' is the function name, 'NA' is the number of arguments of the function and 'FB' is the function body. Functions expect their n arguments loaded in the accumulators 1 to n; a function body is a list of S-expressions which are either labels (if atomic) or instructions.

We now come to describe the nine instructions that LCom0 makes use of (we use **C**[n] to stand for "contents of accumulator n"):

(JRST 0 L)	is an unconditional jump to label L in the current function;
(JUMPE n L)	causes a jump to L in the current function if contents of accumulator n is zero;
(JUMPN n L)	causes a jump to L in the current function if contents of accumulator n is nonzero;
(MOVEI n (QUOTE x))	contents of accumulator n (**C**[n])is set to the coding of S-expression x;
(MOVE n m P)	**C**[n] is set to **C**[**C**[!P] + m]
(PUSH P n)	increments the stack pointer (acc !P) by one and puts **C**[n] on the stack;
(SUB P (C 0 0 n n))	decrements the stack pointer (acc !P) by n;
(CALL n FN)	current routine is suspended and control passes to function in program with name FN (which presumably has n parameters) after incrementing stack pointer by one; If FN is a

	standard function it will restore the value of stack pointer before entry and leave its result in accumulator 1;
(POPJ P)	return from current function to instruction after the one that CALLed the current fun. (stack pointer is decremented by 1);

The particular accumulator numbered !P (referred to by the name 'P' in the above instructions) is used as a stack pointer. Note that we do not worry about stack overflow since we are not assuming finiteness of memory. Also note that, since the arguments of a function are passed in the low accumulators, the maximum number of arguments for a function is less than !P.

9.4. LCom0 LAP - Formal Description:

9.4.1. States and functions on states:

The notion of 'state', in the following semantics, is intended to reflect the correspondence between word addresses and contents - not as a function but as an association list. More specifically, a **state** will be an A-list of pairs (n∘x) where n is an address and x is the coding (by function 'code') of an S-expression; a property of these A-lists is that the pairs are in order of increasing address.

The following axiom gives functions for changing and interrogating states and also other properties of the memory:

```
**AXIOM TL1:

    get ≡ [λx st. tl(assoc(x,st))],

    set ≡ [μG.[λx y st. null(st)→ (x·y)·NIL, (hd(hd(st))=x)→ (x·y)·tl(st),
                          (hd(hd(st))>x)→ (x·y)·st, hd(st)·G(x,y,tl(st))]],

    putargs ≡ [λa st. length(a)>mna→⊥,putargx(length(a),rev(a),st)],
    putargx ≡ [μG.[λn x st. Z(n)→ st, set(n,code(hd(x)),G(n-1,tl(x),st))]],

    argsin ≡ [λa st. length(a)>mna→𝔽,argsinx(length(a),rev(a),st)],
    argsinx ≡ [μG.[λn x st. Z(n)→ 𝕋, (get(n,st)=hd(x))→G(n-1,tl(x),st),𝔽]],

    PDL > !P ≡ 𝕋,
    !P > mna ≡ 𝕋,
    mna ≥ 2 ≡ 𝕋,

    code(NIL)≡0,
    ∀x. dec(code(x))≡x
```

The function 'get' is for interrogating the memory and takes one argument - an address; the function 'set' is used for putting information in the memory and its arguments are an address and a value. 'putargs' puts a list of arguments (values) into the accumulators starting at number 1; 'argsin' testifies that a list of arguments is already contained in the accumulators (starting at 1). The constant 'mna' denotes the maximum number of arguments for functions while '!P' is the address of the stack pointer (an index register).

9.4.2. LAP Functions and operations on them:

We have characterised a LAP program as a collection of LAP functions and so a program is a sort of environment in which to execute function calls. Actually in the axiomatisation a program will be an A-list from which we can extract function bodies and check numbers of parameters. The function ‘body’ does just this:

**AXIOM TL2:

```
||      body ≡ [λfn P n. (n≥hd(tl(assoc(fn,P))))→tl(tl(assoc(fn,P))), ⊥]
```

Now when we are dealing with a LAP function we want to consider it simply a sequence of instructions and labels. Hence the LCF functions we define in the axiom below are applicable to all groups of instructions and labels:

**AXIOM TL3:

```
||    INST ≡ [μH. [λg n. atom(hd(g))→H(tl(g),n),Z(n)→hd(g),H(tl(g),n-1)]],
||
||    loc ≡ [μH. [λx g. atom(hd(g))→ (hd(g)=x)→0,H(x,tl(g)), H(x,tl(g))+1]],
||
||    GL ≡ [μH. [λg. null(g)→0, atom(hd(g))→ H(tl(g)), H(tl(g))+1]],
||
||    complete ≡ [λg exc. comp2(g,labs(g)&exc)],
||
||    comp2 ≡ [μH.[λg labs. null(g)→𝕋, atom(hd(g))→H(tl(g),labs),
||                  isJUMP(hd(hd(g))) →
||                                mem(hd(tl(tl(hd(g)))),labs)→H(tl(g),labs),𝔽,
||                                H(tl(g),labs)]],
||
||    labs ≡ [μH.[λg. null(g)→NIL,atom(hd(g)) → hd(g)·H(tl(g)), H(tl(g))]]
```

If **X** is a group of instructions and labels then 'INST(**X**, **n**)' will pick out the **n**-th instruction, 'GL(**X**)' will compute the number of instructions in **X**, 'labs(**X**)' will list all the labels in **X** and 'loc(**L**, **X**)' will compute the number of instructions that precede label **L** in **X**.

The predicate 'complete' is used to indicate whether all labels referred to by 'JUMP instructions' (in a group of instructions and labels) are also in the group or in a list (of labels) which is the other parameter.

9.4.3. Interpreting LAP.

The highest level function of the semantics of LAP will be called 'lap' and will take three arguments; 'lap(**F**, **L**, **P**)' is to be the result of executing, inside program **P**, the function **F** with actual parameter list **L** (of S-expressions).

The next level of interpreting-function must manage the 'flow of control' within function bodies - or, more generally, within arbitrary sequences of labels and instructions. Defined below is a function 'exec' which gives the effect (on a state) of executing a group of orders (from some point onwards) in the context of some program. More particularly, 'exec(**G**, **P**, **c**, **st**)' will be the (possibly flagged) state produced by executing **G** (a group of instructions) from **P** (a program) starting at the **c**-th instruction of **G** and with initial state **st**. States are flagged while executing a group of orders to indicate that an '**exit**' instruction such as '(POPJ P)' has been encountered. This flagging (accomplished by pairing 'T' with the state) is undone when control gets back to the instruction that 'called' the function being executed.

Naturally, the function 'exec' is written in terms of the meanings of individual orders. Now, since no instruction may do more than affect the memory and cause a transfer of control to its label, we are able to specify the semantics of individual orders by means of two LCF functions - 'NST' (New STate) and 'TOC' (Transfer Of Control). To define these explicitly would be to give the semantics of the entire instruction set so we just axiomatise it for the particular cases we are interested in.

'TOC' is an LCF function of type $(D_{ind} \rightarrow (D_{ind} \rightarrow D_{tr}))$ and 'TOC(**I**, **st**)' indicates whether **I** (a jump instruction) should cause a transfer of control if executed in **st** (an unflagged state). Since it is only applicable for jump instructions, there is a predicate 'isJUMP' whose value is axiomatised for each of the nine instructions we consider.

'NST(**I**, **e**, **P**, **st**)' gives the new state after executing instruction **I** in state **st** and in the context of program **P**; '**e**' has the same type as 'exec' and is used to interpret a function if one is called by **I**. This 'extraneous' parameter is required because we want to define 'exec' in such a way that it is not mutually recursive with 'NST' which will only be partially specified.

Here then are the definitions for 'lap' and 'exec':

```
**AXIOM TL4:

    lap ≡ [λfn args P. dec (get(1,tl(exec(body(fn,P,length(args)), P, 0,
                                          putargs(args,set(!P,PDL,NIL))) ))) ],

    exec ≡ [μH. [λg P c st.
                 (c=GL(g)) → st,
                 (hd(st)=T) → st,
                     [λz. H(g, P,
                          isJUMP(hd(z)) → TOC(z,st)→loc(hd(tl(tl(z))),g),(c+1),
                                                    (c+1),
                          NST(z,H,P,st))] (INST(g,c))]],

    isJUMP(JRST)≡𝕋,     isJUMP(JUMPE)≡𝕋     isJUMP(JUMPN)≡𝕋,
    isJUMP(MOVE)≡𝔽,     isJUMP(MOVEI)≡𝔽,    isJUMP(SUB)≡𝔽,
    isJUMP(PUSH)≡𝔽,     isJUMP(POPJ)≡𝔽,     isJUMP(CALL)≡𝔽 .
```

Refer to Figures 9.6 and 9.7 for the specification of the functions 'NST' and 'TOC', as appropriate, for each of the nine instructions that we consider in our treatment of LAP.

9.5. Towards a Theory of LAP.

The aim of this part of the thesis is to prove the correctness of LCom0 and we do not have time to consider developing even an elementary theory of the language LAP. However, we have given an axiomatic framework for defining most aspects of the language, we have been forced to prove some basic lemmas and so we actually have the beginnings of a theory.

```
**AXIOM TL5:

    ∀x. isname(x)⇒  TOC({JRST 0 x}) ≡ [λst. 𝕋],
    ∀x. isname(x)⇒  NST({JRST 0 x})
                        ≡ [λe fl st. st],

    ∀x. isname(x)⇒  TOC({JUMPE 1 x}) ≡ [λst. Z(get(1,st))],
    ∀x. isname(x)⇒  NST({JUMPE 1 x})
                        ≡ [λe fl st. st],

    ∀x. isname(x)⇒  TOC({JUMPN 1 x}) ≡ [λst. Z(get(1,st))→𝔽,𝕋],
    ∀x. isname(x)⇒  NST({JUMPN 1 x})
                        ≡ [λe fl st. st],

    ∀x. NST({MOVEI 1 {QUOTE x}})
                        ≡ [λe fl st. NIL∘set(1,code(x),st)],

    ∀x y. isint(y)⇒  NST({MOVE x y P})
            ≡ [λe fl st. (0>x)→⊥, (x>mna)→⊥,set(x,(get(P,st)+y),st)],

    ∀x. NST({SUB P {C 0 0 x x}}) ≡ [λe fl st. set(P,get(P,st)-x,st)],

    ∀x. (x>0)⇒  NST({PUSH P x})
            ≡ [λe fl st.[λz. set(P,z+1,set(z,get(x,st),st)](get(P,st))],

    NST({POPJ P}) ≡ [λe fl st. T∘st]
```

Figure 9.6 - Partial Semantics of 8 Lap Instructions.

**AXIOM TL6:

```
∀x y. NST({CALL x y})
        ≡ [λe fl st. (x>mna) → ⊥,
                             isBF(y) → set(1,callBF(y,x,st),st),
                             tl(e(body(y,fl,x),fl,0,NIL·st))],

callBF(CAR) ≡ [λn st. (n≥1)→hd(get(1,st)),⊥ ],

callBF(CDR) ≡ [λn st. (n≥1)→tl(get(1,st)),⊥ ],

callBF(CONS) ≡ [λn st. (n≥2)→ get(1,st)·get(2,st), ⊥],

callBF(LIST) ≡ [λn st. [μf.[λi. i>n → NIL, get(i,st)·f(i+1)]](1)],

callBF(ATOM) ≡ [λn st. (n≥1)→atom(get(1,st))→code(T),code(NIL),⊥ ],

callBF(EQUAL) ≡ [λn st. (n≥2)→ ((get(1,st)=get(2,st))→code(T),code(NIL)), ⊥],


callBF(PLUS) ≡ [λn st. (n≥2)→ code(dec(get(1,st))+dec(get(2,st))), ⊥],

callBF(TIMES) ≡ [λn st. (n≥2)→ code(dec(get(1,st))*dec(get(2,st))), ⊥],

callBF(MINUS) ≡ [λn st. (n≥1)→code(mns(dec(get(1,st)))),⊥],

callBF(GENSYM) ≡ [λn st. (n≥1)→code(gensym(dec(get(1,st)))),⊥ ],

callBF(NUMBERP) ≡ [λn st. (n≥1)→isint(dec(get(1,st)))→code(T),code(NIL),⊥ ],

callBF(GREATERP) ≡ [λn st. (n≥2)→
                ((dec(get(1,st))>dec(get(2,st)))→code(T),code(NIL)),⊥ ]
```

Figure 9.7 - Partial Semantics of the CALL Instruction.

There are quite obvious strictness results for the various functions and we point out that they are proved, but the important aspect of the behaviour of the various functions we have introduced is relative to their effects on groups appended together. We give some of these

$$\vdash\ \forall x\ y.\ \mathrm{labs}(x\&y) \equiv \mathrm{labs}(x)\&\mathrm{labs}(y)$$

$$\vdash\ \forall x\ y.\ \mathrm{GL}(x\&y) \equiv \mathrm{GL}(x) + \mathrm{GL}(y)$$

$$\begin{aligned}\vdash\ \forall x\ y\ c.\ \mathrm{INST}(x\&y,c) \equiv (c\geq\mathrm{GL}(x)) \rightarrow\ & \mathrm{INST}(y,\ c-\mathrm{GL}(x)),\\ & \partial(y) \rightarrow \mathrm{INST}(x,c),\ \bot\end{aligned}$$

$$\mathrm{complete}(X,L)\equiv\mathbb{T},\ \mathrm{complete}(Y,L)\equiv\mathbb{T}\ \vdash\ \mathrm{complete}(X\&Y,L)\equiv\mathbb{T}$$

$$\begin{aligned}\vdash\ \forall L\ x\ y.\ \mathrm{loc}(L,x\&y) \equiv \mathrm{mem}(x,\mathrm{labs}(x)) \rightarrow\ & \partial(y)\rightarrow\mathrm{loc}(L,x),\bot,\\ & \mathrm{loc}(L,y)+\mathrm{GL}(x)\end{aligned}$$

By far the most important result that was proved for LAP itself gave the effect of executing two groups of instructions joined together in terms of executing them sequentially.

$$\begin{aligned}&\forall x.\ \mathrm{mem}(x,\mathrm{labs}(G1))\Rightarrow\ \mathrm{mem}(x,\mathrm{labs}(G2))\equiv\mathbb{F},\\ &\qquad \mathrm{complete}(G1,\mathrm{NIL})\equiv\mathbb{T},\ \ \mathrm{complete}(G2,\mathrm{NIL})\equiv\mathbb{T}\\ &\qquad\quad \vdash\ \forall P\ c\ st.\ \mathrm{exec}(G1\&G2,P,c,st)\\ &\qquad\qquad\quad \equiv (c\geq\mathrm{GL}(G1))\rightarrow\ \mathrm{exec}(G2,P,c-\mathrm{GL}(G1),st),\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathrm{exec}(G2,P,0,\mathrm{exec}(G1,P,c,st))\end{aligned}$$

which has the most important corollary that under certain suitable conditions on G1 and G2:

$$exec(G1\&G2,P,0,st) \equiv exec(G2,P,0,exec(G1,P,0,st))$$

We note that the proof of this result required about 300 steps of LCF proof. That is, of course, after certain simple and general theorems are proved about 'complete', 'labs' etc.

CHAPTER 10

Compiler Correctness (II) - Outline of a Proof

Having axiomatised the source and target languages of the compiler, we turn to the compiler itself.

10.1. The Compiler:

We start by exhibiting the compiler itself; Figure 10.1 (next three pages) gives the m-expression form of this 'LISP Function' which (via interpretation) maps S-expressions which are LISP Functions into other S-expressions which are LAP Functions.

In order to talk about the S-expression form of the compiler we must introduce axioms to give the names to the bodies of the various functions.

```
**AXIOM CO1
      Sappn2 ≡  (LAMBDA (X Y) (CONS (CAR Y) (APPEND X (CDR Y))))
    etc.
```

The S-expressions so introduced are Sappn2, Scomp, Sprup, Smkpush, Sloadac, Scomplis, Scompexp, Scomcond, Scombool and Scompandor.

```
appn2[x;y] = cons[car[y];append[x;cdr[y]]]


comp[fn;vars;exp] =
     λ[[n]; append[append[mkpush[n;1];
                         compexp[exp;minus[n];prup[vars;1];
                                gensym[fn]]];
                   list[list[SUB;P;list[C;0;0;n;n;]];
                       list[POPJ;P];NIL]]
     [length[vars]]


prup[vars;n] =
     [null[vars] → NIL;
      T → cons[cons[car[vars];n]; prup[cdr[vars];plus[n;1]]]]


mkpush[n;m] =
     [greaterp[m,n] → NIL;
      T → cons[list[PUSH;P;m]; mkpush[n;plus[m;1]]]]


loadac[n;k] =
     [greaterp[n;0] → NIL;
      T → cons[list[MOVE;k;n;P]; loadac[plus[n;1];plus[k;1]]]]


complis[u;m;vpr;nl] =
     [null[u] → cons[nl;NIL];
      T → λ[[x];appn2[cdr[x];
                     appn2[ ((PUSH P 1));
                           complis[cdr[u];difference[m;1];
                                  vpr;car[x]]]]]
            [compexp[car[u];m;vpr;nl]]
```

Figure 10.1a - The LISP Functions that Make up LCom0

```
compexp[exp;m;vpr;nl] =
     [or[null[exp];equal[exp;T]] → list[nl;list[MOVEI;1;list[QUOTE;exp]]];
      atom[exp] → list[nl;list[MOVE;1;plus[m;cdr[assoc[exp;vpr]]];P]];
      or[equal[car[exp];AND];equal[car[exp];OR];equal[car[exp];NOT]] →
         append[combool[exp;m;nl;NIL;vpr;gensym[gensym[nl]]];
               list[ (MOVEI 1 (QUOTE T));
                    list[JRST; gensym[nl]];
                    nl; (MOVEI 1 (QUOTE NIL)); gensym[nl]]];
      equal[car[exp];COND] → comcond[cdr[exp];m;nl;vpr;gensym[nl]];
      equal[car[exp];QUOTE] → list[nl;list[MOVEI;1;exp]];
      atom[car[exp]] →
         λ[[n]; append[complis[cdr[exp];m;vpr;nl];
                        append[loadac[difference[1;n];1];
                                 list[list[SUB;P;list[C;0;0;n;n]];
                                     list[CALL;n;list[E;car[exp]]]]]]]
         [length[cdr[exp]]];
     equal[car[car[exp]];LAMBDA] →
         λ[[n;x];append[appn2[cdr[x];
                                compexp[car[cdr[cdr[car[exp]]]];
                                          difference[m;n];
                                          append[prup[car[cdr[car[exp]]];
                                                       difference[1;m]];
                                                 vpr];
                                          car[x]]];
                        list[list[SUB;P;list[C;0;0;n;n]]]]]
         [length[cdr[exp]]; complis[cdr[exp];m;vpr;nl]]
```

Figure 10.1b - The LISP Functions that Make up LCom0 (ctd.)

```
comcond[u;m;l;vpr;nl] =
     [null[u] → list[nl;l];
      T ≡ λ[[x]; λ[[y]; appn2[cdr[x];
                          appn2[cdr[y];
                                 appn2[list[list[JRST;l];nl];
                                        comcond[cdr[u];m;l;vpr;car[y]]]]]]
                 [compexp[car[cdr[car[u]]];m;vpr;car[x]]]
          [combool[car[car[u]];m;nl;NIL;vpr;gensym[nl]]]]

combool[p;m;l;flg;vpr;nl] =
     [atom[p] → append[compexp[p;m;vpr;nl];
                    list[list[ [flg → JUMPN; T → JUMPE];1;l]]];
      equal[car[p];AND] → [not[flg] → compandor[cdr[p];m;l;NIL;vpr;nl];
                           T → append[compandor[cdr[p];m;nl;NIL;vpr;gensym[nl]];
                                      list[list[JRST;l];nl]]];
      equal[car[p];OR] → [flg → compandor[cdr[p];m;l;T;vpr;nl];
                          T → append[compandor[cdr[p];m;nl;T;vpr;gensym[nl]];
                                     list[list[JRST;l];nl]]];
      equal[car[p];NOT] → combool[car[cdr[p]];not[flg];vpr;nl];
      T → append[compexp[p;m;vpr;nl];
              list[list[[flg → JUMPN; T → JUMPE];1;l]]]]

compandor[u;m;l;flg;vpr;nl] =
     [null[u] → list[nl];
      T → λ[[x]; appn2[cdr[x];compandor[cdr[u];m;l;flg;vpr;car[x]]]]
              [combool[car[u];m;l;flg;vpr;nl]]]
```

Figure 10.1c - The LISP Functions that Make up LCom0 (ctd.)

10.1.1. Some Slight Changes

Close comparison of this compiler with the original will reveal that there are small differences. We have already indicated that, in LCom0 LISP, the function GENSYM takes one argument (usually the name it generated last time it was invoked) instead of no arguments (as in LISP 1.5). This change in the language was compensated by a suitable change in the compiler: each function that could generate labels internally acquired an extra parameter - namely, the next label to be used; also each of these functions gave as result a pair of next-label-to-be-generated and a list-of-instructions.

Finally, there is some slight saving in the number of subsidiary functions required. For example, LESSP is avoided by changing the program to use GREATERP.

10.1.2. Predicate 'CFD' - Compiler Functions Defined

Having available the S-expression forms of all the compiler functions, we now introduce an axiom to define a predicate (on lists) which can testify to all the LISP Functions used (directly or indirectly) by LCom0 being in a function list:

```
**AXIOM CO2:
          CFD ≡ [λfl. BFD(fl) →
                 tl(assoc(PRUP,fl))=Sprup →
                  tl(assoc(MKPUSH,fl))=Smkpush →
                   tl(assoc(LOADAC,fl))=Sloadac →
                    tl(assoc(APPN2,fl))=Sappn2 →
                     tl(assoc(COMP,fl))=Scomp →
                      tl(assoc(COMPEXP,fl))=Scompexp →
tl(assoc(COMPLIS,fl))=Scomplis →
 tl(assoc(COMCOND,fl))=Scomcond →
  tl(assoc(COMBOOL,fl))=Scombool →
   tl(assoc(COMPANDOR,fl))=Scompandor,
                  𝔽,𝔽,𝔽,𝔽,𝔽, 𝔽,𝔽,𝔽,𝔽,𝔽]
```

10.2. Meaning of the Compiler:

Figure 10.2 (next three pages) gives the meaning functions that the compiler LISP Functions induce under interpretation. Figure 10.3 contains theorems which explicate the definitions of 'compexp' and 'combool' which are the hardest to follow. We shall therefore consider the LCF function 'comp' to be the compiling algorithm of LCom0. The purpose of introducing the meaning functions is to factor the whole proof of correctness of the compiler into two substantial but independent parts:

i) the correctness of the S-expression form of LCom0 relative to the compiling algorithm;

ii) the correctness of the compiling algorithm.

The technical statement of the first subproblem is simply:

CFD(FL)≡𝕋 ⊢ ∀f v e. apply(COMP,{f v e},NIL,FL) ≡ comp(f,v,e)

which we arrive at via the family of lemmas:

```
CFD(FL)≡𝕋 ⊢ ∀e m vpr nl vb. apply(COMPEXP,{e m vpr nl},vb,FL)
                         ≡ islist(vb) → compexp(u,m,vpr,nl), ⊥
```

```
**AXIOM C03:

    comp ≡ [λf v e.[λn.(mkpush(n,1)
                     &tl(compexp(e,mns(n),prup(v,1),gensym(f))))
                      & { {SUB P {C 0 0 n n}} {POPJ F} NIL}]length(v)],

    compexp ≡ [μG.[λexp m vpr nl.
              (null(exp)→𝕋,(exp=T)→𝕋,isint(exp)) →
                                    {nl {MOVEI 1 {QUOTE exp}}},
              atom(exp) → {nl {MOVE 1 m+tl(assoc(exp,vpr)) P}},
              ((hd(exp)=AND)→𝕋,(hd(exp)=OR)→𝕋,(hd(exp)=NOT)) →
                    comboolF(G)(exp,m,nl,NIL,vpr,gensym(gensym(nl)))
                          & { {MOVEI 1 {QUOTE T}} {JRST 0 gensym(nl)}
                                nl  {MOVEI 1 {QUOTE NIL}}  gensym(nl)},
              (hd(exp)=COND) →
                  comcondF(G,comboolF(G)) (tl(exp),m,nl,vpr,gensym(nl)),
              (hd(exp)=QUOTE) → {nl {MOVEI 1 exp}},
              atom(hd(exp)) → complisF(G)(tl(exp),m,vpr,nl)
                 & [λn.loadac(1-n,1) & { {SUB P {C 0 0 n n}}
                                {CALL n {E hd(exp)}}}](length(tl(exp))),
              (hd(hd(exp))=LAMBDA) →
                 [λn x vpr2. appn2(tl(x),G(hd(tl(tl(hd(exp)))),m-n,vpr2,hd(x)))
                       & {{SUB P {C 0 0 n n}}}](length(tl(exp)),
                                             complisF(G)(tl(exp),m,vpr,nl),
                                             vpr & prup(hd(tl(hd(exp))),1-m)),
              ⊥]],

   complisF ≡ [λce. [μH.[λu m vpr nl. null(u)→ {nl},
              [λx. appn2(tl(x), appn2( { {PUSH P 1}},
                     H(tl(u),m-1,vpr,hd(x))))](ce(hd(u),m,vpr,nl))]]],
```

Figure 10.2a 'comp' - the meaning of 'COMP.

```
comboolF ≡ [λce. [μH.[λp m l flg vpr nl.
          atom(p) → ce(p,m,vpr,nl)
                     & { {(null(flg)→JUMPE,JUMPN) 1 l}},
          (hd(p)=AND) →
             null(flg) → compandorF(H)(tl(p),m,l,NIL,vpr,nl),
                       compandorF(H)(tl(p),m,nl,NIL,vpr,gensym(nl))
                            & { {JRST 0 l} nl},
          (hd(p)=OR) →
             null(flg) → compandorF(H)(tl(p),m,nl,T,vpr,gensym(nl))
                           & { {JRST 0 l} nl},
                       compandorF(H)(tl(p),m,l,T,vpr,nl),
          (hd(p)=NOT) → H(hd(tl(p)),m,l,(null(flg)→T,NIL),vpr,nl),
                        ce(p,m,vpr,nl) & { {(null(flg)→JUMPE,JUMPN) 1 l}}]]],


compandorF ≡ [λcb. [μF.[λu m l flg vpr nl. null(u)→ {nl},
                    [λx. appn2(tl(x),F(tl(u),m,l,flg,vpr,hd(x)))]
                               (cb(hd(u),m,l,flg,vpr,nl))]]],

comcondF ≡ [λce cb. [μH.[λu m l vpr nl. null(u) → {nl l},
             [λx. [λy. appn2(tl(x),
                           appn2(tl(y),
                                 appn2( { {JRST 0 l} nl},
                                        H(tl(u),m,l,vpr,hd(y)))))]
                   (ce(hd(tl(hd(u))),m,vpr,hd(x)))]
             (cb(hd(hd(u)),m,nl,NIL,vpr,gensym(nl)))]]],
```

Figure 10.2b - Auxiliary Functions for 'comp'.

```
complis ≡ complisF(compexp),

combool ≡ comboolF(compexp),

compandor ≡ compandorF(combool),

comcond ≡ comcondF(compexp, combool)


appn2 ≡ [λx y. hd(y) · (x & tl(y)) ],

prup ≡ [μG.[λv n. null(v)→NIL, (hd(v)·n) · G(tl(v),n+1)]],

mkpush ≡ [μG.[λn m. (m>n)→NIL, {PUSH P m} · G(n,m+1)]],

loadac ≡ [μG.[λn k. (n>0)→NIL, {MOVE k n P} · G(n+1,k+1)]]
```

Figure 10.2c - Auxiliary Functions for 'comp'.

```
compexp ≡ [μG.[λexp m vpr nl.
          (null(exp)→𝕋,(exp=T)→𝕋,isint(exp)) →
                                {nl {MOVEI 1 {QUOTE exp}}},
          atom(exp) → {nl {MOVE 1 m+tl(assoc(exp,vpr)) P}},
          ((hd(exp)=AND)→𝕋,(hd(exp)=OR)→𝕋,(hd(exp)=NOT)) →
             combool(exp,m,nl,NIL,vpr,gensym(gensym(nl)))
             & { {MOVEI 1 {QUOTE T}} {JRST 0 gensym(nl)}
                 nl  {MOVEI 1 {QUOTE NIL}}  gensym(nl)},
          (hd(exp)=COND) →
               comcond(tl(exp),m,nl,vpr,gensym(nl)),
          (hd(exp)=QUOTE) → {nl {MOVEI 1 exp}},
          atom(hd(exp)) → complis(tl(exp),m,vpr,nl)
             & [λn.loadac(1-n,1) & { {SUB P {C 0 0 n n}}
                     {CALL n {E hd(exp)}}}](length(tl(exp))),
          (hd(hd(exp))=LAMBDA) →
             [λn x vpr2. appn2(tl(x),G(hd(tl(tl(hd(exp)))),m-n,vpr2,hd(x)))
                   & {{SUB P {C 0 0 n n}}}](length(tl(exp)),
                              complis(tl(exp),m,vpr,nl),
                                 vpr & prup(hd(tl(hd(exp))),1-m)),
           ⊥]],


combool ≡ [μH.[λp m l flg vpr nl.
          atom(p) → compexp(p,m,vpr,nl)
                  & { {(null(flg)→JUMPE,JUMPN) 1 l}},
          (hd(p)=AND) →
             null(flg) → compandor(tl(p),m,l,NIL,vpr,nl),
                      compandor(tl(p),m,nl,NIL,vpr,gensym(nl))
                            & { {JRST 0 l} nl},
          (hd(p)=OR) →
             null(flg) → compandor(tl(p),m,nl,{T,vpr,gensym(nl))
                            & { {JRST 0 l} nl},
                      compandor(tl(p),m,l,T,vpr,nl),
          (hd(p)=NOT) → H(hd(tl(p)),m,l,(null(flg)→T,NIL),vpr,nl),
          compexp(p,m,vpr,nl)& { {(null(flg)→JUMPE,JUMPN) 1 l}}]]],
```

Figure 10.3 - Theorems Explicating 'compexp' and 'combool'.

$$\mathrm{CFD}(\mathrm{FL}) \equiv \mathrm{TT} \;\vdash\; \forall u\ m\ l\ vpr\ nl\ vb.\ \mathrm{apply}(\mathrm{COMCOND},\ulcorner u\ m\ l\ vpr\ nl\urcorner,vb,\mathrm{FL}) \\ \equiv \mathrm{islist}(vb) \rightarrow \mathrm{comcond}(u,m,l,vpr,nl),\ \bot$$

etc.

The appropriate attack on these subproblems is by means of the techniques described in the context of Pure LISP (see Chapter 7). We must prove the family of lemmas simultaneously using induction on the the structure of the expression being compiled. The proof will clearly be long and for this reason alone we would find difficulty in establishing the results. We estimate that it would be comparable in size to that half of the interpreter proof that was done on the machine.

10.3. **Properties of the Compiler Functions.**

Having extracted meanings for the various compiler functions as terms of LCF, we must proceed to prove various theorems about their behaviour. The most important one is treated in the next section: that the compiling functions produce 'correct LAP code. In this section we present some useful but much simpler lemmas about the LCF functions 'comp', 'compandor' etc.

Attached to some of the lemmas there are provisos that the arguments given to a function are well formed. We refer the reader to chapter 9 for the discussion of well-formedness of LISP expressions.

Several of the functions take a parameter which will call a **variable position record** (vpr). A **vpr** is an A-list which associates variables with integers used in the computation of stack positions for variables. The predicate 'isvpr(**v**,**n**)' checks that **v** is a vpr, that the integers are in descending order and are positive but less than **n**. The function 'vprvars' builds a list of all the variables mentioned in a vpr.

$$\begin{array}{ll} \mathrm{isvpr} \equiv [\mu G.[\lambda v\ n. & \mathrm{null}(v) \rightarrow (n \geq 1), \\ & \mathrm{tl}(\mathrm{hd}(v)) \geq n \rightarrow \mathbb{F}, \\ & \mathrm{isname}(\mathrm{hd}(\mathrm{hd}(v))) \rightarrow G(\mathrm{tl}(v),\mathrm{tl}(\mathrm{hd}(v))),\ \mathbb{F}\ ]] \end{array}$$

$$\mathrm{vprvars} \equiv [\mu G.\ [\lambda x.\ \mathrm{null}(x) \rightarrow \mathrm{NIL},\ \mathrm{hd}(\mathrm{hd}(x)) \cdot G(\mathrm{tl}(x))]]$$

We observe that 'compexp', 'complis', 'combool', 'compandor', 'comcond' are all strict in their first and last arguments and that 'compexp' is strict in all its arguments.

10.3.1. Totality

The result we suggest in this subsection is that each of the compiler functions terminates with a list (of instructions) provided only that its arguments is well-formed. Formal statements of two instances of this result are:-

$$\begin{array}{l} \mathrm{iswfe}(e,\mathrm{vprvars}(vpr)) \equiv \mathbb{T}, \\ \mathrm{isvpr}(vpr,\mathrm{mns}(m)) \equiv \mathbb{T}, \\ \mathrm{isname}(nl) \equiv \mathbb{T} \\ \qquad \vdash\ \mathrm{islist}(\mathrm{compexp}(e,m,vpr,nl)) \equiv \mathbb{T} \end{array}$$

and

$\text{iswfl}(p,\text{vprvars}(vpr)) \equiv \mathbb{T},$
$\text{isvpr}(vpr,\text{mns}(m)) \equiv \mathbb{T},$
$\text{isname}(l) \equiv \mathbb{T},$
$(flg{=}\text{NIL}) \rightarrow \mathbb{T},\ (flg{=}\text{T}) \equiv \mathbb{T},$
$\text{isname}(nl) \equiv \mathbb{T}$
$\quad \vdash\ \text{islist}(\text{combool}(p,m,l,flg,vpr,nl)) \equiv \mathbb{T}\ .$

By instantiating the first of these two lemmas appropriately we get:

$\text{iswfe}(\ \{\text{LAMBDA}\ v\ e\},\ \text{NIL}) \equiv \mathbb{T},$
$\text{isname}(f) \equiv \mathbb{T}$
$\quad \vdash\ \text{islist}(\text{comp}(f,v,e)) \equiv \mathbb{T}$

10.3.2. Completeness

We next suggest some results which say that the bodies of code produced by compiler functions are complete in the sense that they contain no jumps to 'undefined' labels. Take, for example, 'combool':

$\text{iswfl}(p,\text{vprvars}(vpr)) \equiv \mathbb{T},$
$\text{isvpr}(vpr,\text{mns}(m)) \equiv \mathbb{T},$
$(flg{=}\text{NIL}) \rightarrow \mathbb{T},\ (flg{=}\text{T}) \equiv \mathbb{T},$
$\text{discr}(nl) > \text{discr}(l) \equiv \mathbb{T}$
$\quad \vdash\ \text{complete}(\text{tl}(\text{combool}(p,m,l,flg,vpr,nl)),\ \{l\}\ ) \equiv \mathbb{T}\ .$

The corresponding theorem for 'comp' is:

$\text{iswfe}(\ \{\text{LAMBDA}\ v\ e\}\ ) \equiv \mathbb{T},$
$\text{isname}(f) \equiv \mathbb{T}$
$\quad \vdash\ \text{complete}(\text{comp}(f,v,e),\ \text{NIL}) \equiv \mathbb{T}\ .$

10.3.3. Distribution of Labels

When we come to prove correctness of the compiler functions we will need lemmas which declare that in bodies of code produced by the compiler functions, labels are declared only once. This requirement is fulfilled by some theorems which describe the orderly placing of labels. For example, we state the one for 'comcond':

$$\begin{array}{l}
\mathrm{iswf1}(u,\mathrm{vprvars}(vpr)) \equiv \mathbb{T}, \\
\mathrm{isvpr}(vpr,\mathrm{mns}(m)) \equiv \mathbb{T}, \\
\mathrm{discr}(nl) > \mathrm{discr}(l) \equiv \mathbb{T}, \\
X \equiv \mathrm{comcond}(u,m,l,vpr,nl), \\
\mathrm{mem}(y, \mathrm{labs}(\mathrm{tl}(X))) \equiv \mathbb{T} \\
\qquad \vdash \mathrm{discr}(y) \geq \mathrm{discr}(l) \equiv \mathbb{T}, \\
\qquad\quad \mathrm{discr}(\mathrm{hd}(X)) > \mathrm{discr}(y) \equiv \mathbb{T} \quad .
\end{array}$$

10.4. Statement of Correctness.

Let us now state what our final goal is. We first do so informally as follows:

IF we have a certain function list **FL1** of well-formed LISP Functions

AND we have a function list **FL2** of the compiled forms of those LISP Functions (where compilation is done by running the LISP compiler (LCom0)),

THEN the effect of applying some function **F** to some list **A** of arguments (not too long) is the same whether we use LISP 'apply' in the context of **FL1** or LAP in the context of **FL2**.

That is, we must establish the theorem,

$$\begin{array}{l}
\forall x.\ \partial(\mathrm{hd}(\mathrm{assoc}(x,\mathrm{FL1}))) \Rightarrow \mathrm{iswfe}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1}))) \equiv \mathbb{T}, \\
\forall x.\ \partial(\mathrm{hd}(\mathrm{assoc}(x,\mathrm{FL1}))) \Rightarrow \mathrm{hd}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1}))) \equiv \mathrm{LAMBDA}, \\
\mathrm{CFD}(\mathrm{FL}) \equiv \mathbb{T}, \\
\forall x.\ \partial(\mathrm{hd}(\mathrm{assoc}(x,\mathrm{FL2}))) \Rightarrow \\
\qquad \mathrm{tl}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL2}))) \equiv \mathrm{apply}(\mathrm{COMP},\ \{x\ \ \mathrm{hd}(\mathrm{tl}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1})))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{hd}(\mathrm{tl}(\mathrm{tl}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1}))))) \},\ \mathrm{NIL},\ \mathrm{FL}) \\
\qquad \vdash\ \forall \mathrm{fn}\ \mathrm{args}.\ \mathrm{length}(\mathrm{args}) \le \mathrm{mna} \Rightarrow \\
\qquad\qquad \mathrm{apply}(\mathrm{fn},\mathrm{args},\mathrm{NIL},\mathrm{FL1}) \equiv \mathrm{lap}(\mathrm{fn},\mathrm{args},\mathrm{FL2})
\end{array}$$

10.4.1. Correctness of the Compiling Algorithm

In Section 2 we exhibited the function 'comp' which is the one induced under interpretation by the LISP function 'COMP'. We are thus entitled to simplify the compiler correctness problem by rewriting some of the hypotheses of the above theorem. We will now assume those modified hypotheses for the rest of the chapter, effectively creating constants FL1 and FL2:-

$$\begin{array}{l}
\forall x.\ \partial(\mathrm{hd}(\mathrm{assoc}(x,\mathrm{FL1}))) \Rightarrow \mathrm{iswfe}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1}))) \equiv \mathbb{T}, \\
\forall x.\ \partial(\mathrm{hd}(\mathrm{assoc}(x,\mathrm{FL1}))) \Rightarrow \mathrm{hd}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1}))) \equiv \mathrm{LAMBDA}
\end{array}$$

and

$$\begin{array}{l}
\forall x.\ \partial(\mathrm{hd}(\mathrm{assoc}(x,\mathrm{FL2}))) \Rightarrow \\
\qquad \mathrm{tl}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL2}))) \equiv \mathrm{comp}(x,\ \mathrm{hd}(\mathrm{tl}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1})))), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{hd}(\mathrm{tl}(\mathrm{tl}(\mathrm{tl}(\mathrm{assoc}(x,\mathrm{FL1}))))))
\end{array}$$

The correctness of the compiling algorithm is then just:

$$\vdash \forall \mathrm{fn}\ \mathrm{args}.\ \mathrm{length}(\mathrm{args}) \le \mathrm{mna} \Rightarrow \mathrm{apply}(\mathrm{fn},\mathrm{args},\mathrm{NIL},\mathrm{FL1}) \equiv \mathrm{lap}(\mathrm{fn},\mathrm{args},\mathrm{FL2})$$

10.4.2. The Principal Lemma

Taking the result of the last subsection as our goal, we see that the appropriate principal subgoal is:

```
∀vb st args fn. ∂(vb)⇒ length(args)≤mna⇒ get(!P,st)≥PDL⇒
        dec(get(1,exec(body(fn,FL2,length(args)), FL2, 0, putargs(args,st))))
                     ≡ apply(fn,args,vb,FL1)
```

The main correctness result follows from this one by taking 'vb' to be 'NIL' and 'st' to be 'set(!P,PDL,NIL)' .

10.4.3. Environment Correspondence

At the next level of goals, we will have equations where the LAP interpretation of some expression appears on the left hand side and LISP interpretation of a corresponding expression appears on the right. However, both of these interpreting functions take an environment as a parameter and so we will sometimes need preconditions to the effect that a pair of environments are consistent. We thus define a correspondence function between LISP A-Lists and LAP run-time stacks as follows:

```
stkscorr ≡ [μG.[λvb st vpr m.
                null(vpr) → (get(!P,st)+m ≥ PDL),
                (hd(hd(vb))=hd(hd(vpr)))
                        → (tl(hd(vb))=get(get(!P,st)+m+tl(hd(vpr)),st)
                                      → G(tl(vb),st,tl(vpr),m),  𝔽),
                          𝔽]]
```

One sees that if stkscorr(**vb**,**st**,**vpr**,**m**)≡T then the value of any variable extractable from the run-time stack by means of the function [λx. get(get(!P,**st**)+**m**+tl(assoc(x,**vpr**)),**st**)] is the same as the value which would be extracted from the A-list **vb** by means of the usual function [λx. tl(assoc(x,**vb**))]. Note that this correspondence function is very much tailored to our present purposes of proving LCom0. A more general such predicate might not require that variables appear in exactly the same order in the A-list and the stack; on the other hand, it could require that all of the stack in **st** should correspond to all of **vb** instead of just those variables that are mentioned in **vpr**.

10.4.4. Second Level Subgoals

The secondary lemmas which we must prove and which we list in figures 10.4 to 10.8 relate LISP interpretation in some environment (an A-list) to LAP execution of corresponding code in a corresponding environment (a stack). More particularly, we wish to describe the effects of executing code produced by 'compexp', 'complis', 'comcond', 'combool' and 'compandor' in terms of how the LISP functions 'eval', 'evlis' and 'evcon' operate on the source S-expressions. Note that there are just three effects we wish to capture in lemmas about code execution:

i) What the answer is (usually what register 1 contains);

ii) How the stack pointer is affected;

iii) How the stack contents are affected.

i) Answer:

```
∀exp vb st vpr m lab.
  stkscorr(vb,st,vpr,m)⇒
  iswf1(exp,vprvars(vpr))⇒
  isname(lab)⇒
  dec(get(1,exec(tl(compexp(exp,m,vpr,lab)),FL2,0,st)))
         ≡ eval(exp,vb,FL1)
```

ii) invariance of Stack Pointer:

```
∀exp st vpr m lab.
  iswf1(exp,vprvars(vpr))⇒
  isname(lab)⇒
  get(!P,st)≥PDL⇒
∀st2.
  st2=exec(tl(compexp(exp,m,vpr,lab)),FL2,0,st)⇒
  get(!P,st2) ≡ get(!P,st)
```

iii) Invariance of Stack Contents:

```
∀exp st vpr m lab.
  iswf1(exp,vprvars(vpr))⇒
  isname(lab)⇒
  get(!P,st)≥PDL⇒
∀st2 n.
  st2=exec(tl(compexp(exp,m,vpr,lab)),FL2,0,st)⇒
  n≥PDL⇒
  get(!P,st)>n⇒
  get(n,st2) ≡ get(n,st)
```

Figure 10.4 - Subgoals Describing Effects of 'compexp'.

i) Answer (additions to stack):

```
∀x vb st vpr m lab.
  stkscorr(vb,st,vpr,m)⇒
  iswf2(iswf1,x,vprvars(vpr))⇒
  isname(lab)⇒
  [λst1. [μG. [λn. n≥length(x) → NIL,
                     get(get(!P,st1)-n,st1) · G(n+1)]](1)]
    (exec(tl(complis(x,m,vpr,lab)),FL2,0,st))
       ≡ evlis(x,vb,FL1)
```

ii) Effect on Stack Pointer:

```
∀x st vpr m lab.
  iswf2(iswf1,x,vprvars(vpr))⇒
  isname(lab)⇒
  get(!P,st)≥PDL⇒
∀st2.
  st2=exec(tl(complis(x,m,vpr,lab)),FL2,0,st)⇒
  get(!P,st2) ≡ get(!P,st)+length(x)
```

iii) Invariance of Stack Contents:

```
∀x st vpr m lab.
  iswf2(iswf1,x,vprvars(vpr))⇒
  isname(lab)⇒
  get(!P,st)≥PDL⇒
∀st2 n.
  st2=exec(tl(complis(x,m,vpr,lab)),FL2,0,st)⇒
  n≥PDL⇒
  get(!P,st)>n⇒
  get(n,st2) ≡ get(n,st)
```

Figure 10.5 - Subgoals Describing Effects of 'complis'.

i) Answer:

```
∀x vb st vpr m y1 y2.
  stkscorr(vb,st,vpr,m)⇒
  iswf3(iswf1,x,vprvars(vpr))⇒
  discr(y2)>discr(y1)⇒
  dec(get(1,exec(tl(comcond(x,m,y1,vpr,y2)),FL2,0,st)))
        ≡ evcon(x,vb,FL1)
```

ii) Invariance of Stack Pointer:

```
∀x st vpr m y1 y2.
  iswf3(iswf1,x,vprvars(vpr))⇒
  discr(y2)>discr(y1)⇒
  get(!P,st)≥PDL⇒
∀st2.
  st2=exec(tl(comcond(x,m,y1,vpr,y2)),FL2,0,st)⇒
  get(!P,st2) ≡ get(!P,st)
```

iii) Invariance of Stack Contents:

```
∀x st vpr m y1 y2.
  iswf3(iswf1,x,vprvars(vpr))⇒
  discr(y2)>discr(y1)⇒
  get(!P,st)≥PDL⇒
∀st2 n.
  st2=exec(tl(comcond(x,m,y1,vpr,y2)),FL2,0,st)⇒
  n≥PDL⇒
  get(!P,st)>n⇒
  get(n,st2) ≡ get(n,st)
```

Figure 10.6 - Subgoals Describing Effects of 'comcond'.

```
∀n.  ∀x vb st vpr m y1 y2 flg.
  stkscorr(vb,st,vpr,m)⇒
  iswf1(x,vprvars(vpr))⇒
  discr(y2)>discr(y1)⇒
  (flg=T) → 𝕋, (flg=NIL)⇒
∀xL seq1 seq2.
  xL=tl(combool(x,m,y1,flg,vpr,y2))⇒
  hd(seq2)=y1⇒
  disjoint(labs(seq2),labs(seq1))⇒
  disjoint(labs(seq1&seq2),labs(xL))⇒
  get(!P,st)>n⇒
  (n≥PDL)→𝕋,(n=1)→𝕋,(n=!P)⇒

  get(n,exec(xL&seq1&seq2,FL2,0,st))
       ≡ [λC. get(n,exec(C,FL2,0,st))]
           (null(eval(x,vb,FL1))→(null(flg)→seq2,seq1&seq2),
                                 (null(flg)→seq1&seq2,seq2))
```

Notes:

i) This lemma can be specialised to tell about answer, stack pointer or old stack contents by taking 'n' to be 1, !P or some stack address (an integer between PDL and get(!P,st)).

ii) The predicate 'disjoint' searches for common elements of two lists; It yields 𝔽 if it finds one.

```
  disjoint ≡ [μG. [λx y. null(x) → 𝕋,
                          mem(hd(x),y) → 𝔽, G(tl(x),y)]]
```

Figure 10.7 - Subgoal Describing Effects of 'combool'.

```
      ∀n.  ∀x vb st vpr m y1 y2 flg.
        stkscorr(vb,st,vpr,m)⇒
        iswf1(x,vprvars(vpr))⇒
        discr(y2)>discr(y1)⇒
        (flg=T) → 𝕋, (flg=NIL)⇒
      ∀xL seq1 seq2.
        xL=tl(compandor(x,m,y1,flg,vpr,y2))⇒
        hd(seq2)=y1⇒
        disjoint(labs(seq2),labs(seq1))⇒
        disjoint(labs(seq1&seq2),labs(xL))⇒
        get(!P,st)>n⇒
        (n≥PDL)→𝕋,(n=1)→𝕋,(n=!P)⇒

get(n,exec(xL&seq1&seq2,FL2,0,st))
     ≡ [λC.get(n,exec(C,FL2,0,st))]
        ([μG.[λy.null(y)→seq1&seq2,
                                   null(eval(hd(y),vb,FL1))→(null(flg)→seq2,G(tl(y))),
                                                                 (null(flg)→G(tl(y)),seq2)]](x))
```

Note:

This lemma can be specialised to tell about answer, stack pointer or old stack contents by taking 'n' to be 1, !P or some stack address (an integer between PDL and get(!P,st)).

Figure 10.8 - Subgoal Describing Effects of 'compandor'.

10.4.5. Attacking the Subgoals.

Because 'compexp', 'complis' etc. are all mutually recursive, the subgoals of figures 10.4 to 10.8 are all interdependent. It is thus necessary (but natural) to attack all these subgoals simultaneously. The appropriate tactic will clearly be induction on the structure of all S-expressions being compiled. We do this by using Scott induction on the definition of 'iswf1' which occurs in the relativisations of all the subgoals.

The reader who is unfamiliar with LCF should not be perturbed at the large size of the conjunction of all these formulae; Immediately after the induction tactic is performed the new principal subgoal generated may be split back into manageable pieces.

The reader may also wonder whether the limit on size of core image imposed on the LCF System presents a barrier which can make some proofs effectively impossible to do. The answer to this question is that, in practice, proofs in the system tend to be reasonably well-structured and we can factor such proofs into their main parts and subsidiary parts and then prove subsidiary results in separate core images. More particularly, if a subsidiary part of a proof has N steps, makes reference to J previous steps (hypotheses H_1, H_2, .. H_J) and contains K steps for future use (results R_1, R_2, .. R_K) then as a separate task we may attack $H_1,..,H_J \vdash R_1,..,R_K$. Now if this proof (at most N+J steps long) fits in core and J+K is much less than N (as is usual) we win.

10.5. Feasibility of a Full Compiler Proof

To sum up, we have split the LCom0 correctness problem into four parts. The first two were the developments of axiomatic theories for LISP and LAP and chapter 9 reported on the machine assisted generation of these theories.

The third part of the total problem was the proof that 'compexp' etc. are the functions denoted by the S-expressions COMPEXP etc. This part of the compiler problem was not worked up to a machine checked proof but, for reasons cited above, it was expected to be quite feasible involving two or three man/weeks of effort.

The fourth part was the correctness of the compiling algorithm and we have just presented a natural high level goal structure for achieving the result. It is not thought there would be any conceptual difficulties in forging this plan into a completely formal proof but the time taken to do it must be considerably more than was required for the simple half of the interpreter proof. We estimate that it would be at the very least six man/weeks of effort using the current LCF system.

Thus with upwards of eight man/weeks of effort required it is appropriate to suspend this problem until a more automatic LCF is available.

CHAPTER 11

Second generation LCF System

Since we assert that LCF is a useful (even important) tool for the theory of computation, a major aim in these LISP experiments has been to push the current system to its limits. In many directions the limitations severely handicap the user's ability to specify a proof at a natural level and in a compact way. We present, therefore, in this chapter many suggestions for improvements to the system. These improvements will probably be realised in a second generation system. It must be acknowledged that several of the ideas were developed in conjunction with Richard Weyhrauch and Robin Milner.

11.1. Prior Accomplishments

Although the notion of conditional simplification arose out of earlier work on LCF by Weyhrauch, Milner and Newey, it was implemented for this work. Without that facility the proofs would have been much longer.

The 'PREF tactic' mentioned in Chapter 3 was implemented after the bulk of the Pure LISP proofs were done we credit the current work for its development.

11.2. Proof Generation vs. Proof Checking

The LCF system was conceived as a proof checker which had some ability to help the user generate proofs but the implementation has undergone various mutations which were all intended to make the task of generating easier. Although it still claims to be able to check proofs, the considerable complexity of the more advanced derived deduction rules inevitably diminish confidence in the checking process. In fact, the notions of checking and generating are confused in the system design and inextricably entwined in the actual code. For example, if the user calls for a substitution then LCF generates an appropriate step but the only sense in which anything is checked is that the system checks that the user's prescriptions do indeed generate a step.

Simplification is particularly worrisome in this regard. It is a very complex deduction rule and can change steps so drastically that the user is simply forced to believe that the machine did it all correctly as long as the answer 'looks good'.

What is suggested is that the tasks of generation and checking be realised in completely separate programs. We propose a program which will just check proofs where steps are given in full in a restricted version of the logic and an interactive program which will translate the user's high level notions into a proof that the base checker (the first program) can validate.

It will be most important for the base checker to be simple because we will wish to have confidence in it. As soon as practicable we would want it proved correct. Of course, it would be nice if the interactive program were correct too, but that concern is secondary to its power to produce proofs with a minimum of effort from the user. Since the integrity of the generator is not of great importance, the user should be permitted to supply actual code which can help the system find a proof.

It is clear that we expect the proof generator of the new LCF system (LCF2) to grow up to be an interactive theorem prover for LCF, so more emphasis will be placed on partial decision procedures and automatic selection of deduction rules.

11.3. High Level Command Language

Using the current system is rather reminiscent of using assembly language; the deduction rules correspond to the instructions in that when each command is typed in, one deduction rule is applied. It is clear that, in LCF2, the input language for the base checker will persist in being low level but the language with which we talk to the interactive proof generator should have various features of high level programming languages.

11.3.1. Data Types and Expressions

We propose that there be at least four types - term, wff, step and simpset. It should be possible to have variables of each of these types as well as constants. The 'LABEL' facility of the current system is actually a simple use of variables which have values which are steps. Of course, for convenience of programming, integers should be another data type provided.

There will be many operations on data of the different types, including operators which correspond to many of the deduction rules of the current system. For example, 'abstraction', 'application', 'symmetry', 'transitivity', 'fixed-point', 'substitution' are operators which transform one (or more) items of data into a step.

The notions of expression and assignment follow naturally from these ideas of data types, variables, constants and operators.

11.3.2. Control Structures

It is a trivial consequence of our analogy between LCF command language and conventional programming language that we should incorporate control structures such as procedures, functions, conditional statements, iterative statements, compound statements and blocks. The application of procedures in proof generation is in the binding together as a body many commands that can be then thought of as constituting a recipe for producing proof for some step. The formal parameters of procedures and functions may be of any data type or possibly functions over them. Similarly, an iterative statement would allow some command (or sequence of commands) to be repeatedly executed until some appropriate condition is satisfied. Blocks are useful for delimiting scope of variables.

11.4. Revised Axiom Structure

In the current system one can only present nonlogical axioms to the machine if they have the form of WFFs. Hence, for example, one is prevented from having such notions as $\forall x.\, F(x) \equiv A \vdash A \equiv B$ as an axiom (in that form, at least). This is opposed to the logical axioms of LCF (which are built into the system) such as $s1 \sqsubseteq s2 \vdash t(s1) \sqsubseteq t(s2)$ and to theorems which are allowed to take the form of a sentence. This fact has led many users to adopt the rather unfortunate practice of expressing axiomatic material as unproved theorems (in fact unprovable theorems).

It would appear that the only reason that axioms are not allowed to be sentences is that they are, unlike theorems, made numbered steps in the proof. As such they have a WFF part and a dependency part which must be a list of assumptions. It is proposed that axioms be made to behave more like theorems than regular steps and some of the differences between theorems and steps reduced. In particular, whenever a step expression can appear as an argument to a rule, an axiom or theorem should be permissible.

11.5. Extending the Pure Logic

The two ways of expressing implication in LCF are really rather restrictive. The split arrow (⇒) abbreviation allows relativisation of equations by truth-valued terms only. Also the turnstile (⊢), as used in theorems, can only appear once in a theorem. It was argued in the last section that this turnstile facility, which is also used to express the logical axioms of LCF, should be made available for axiom writing. However, perhaps a more general attack on these expressive weaknesses of the current logic would be more rewarding.

There have been occasional instances where it has proven quite inconvenient to have just the rather simple formula structure we have. A good example is course-of-values induction over the natural numbers, which can best be written for the current system as:

$$\begin{array}{l}\forall y.\ [\mu H.\ [\lambda w.\ Z(w) \rightarrow \mathbb{T}, \\ \qquad\qquad g(pred(w)) \rightarrow H(pred(w)),\bot]](y) \Rightarrow\ g(y)\equiv\mathbb{T}\ .\end{array}$$

Now if we extended our weak notions of implication and universal abstraction we could write

$$\begin{array}{l}g(0)\equiv\mathbb{T},\ \forall y.\ (\forall x.\ y>x\equiv\mathbb{T} \supset g(x)\equiv\mathbb{T}) \supset g(y)\equiv\mathbb{T}\ ) \\ \qquad\qquad \vdash \forall y.\ g(y)\equiv\mathbb{T}\end{array}$$

Inspired by such instances as we have in figures 10.4 to 10.6, we note that in normal situations we can have wffs with identical sequences of prefixes containing cumbersome relativisations. When we conjoin such wffs we would like to only write the prefix sequence once. For example, we would much rather write the goal $\forall x.\ A \supset (B_1, B_2)$ instead of the goal $(\forall x.\ A \Rightarrow B_1), (\forall x.\ A \Rightarrow B_2)$.

The proposal of this section is to have a syntax for Well-Formed Formulae which goes like:

```
<WFF> ::= <equivalence> | <inequivalence>
                        | ∀<varlist> . <WFF>
                        | <WFF> ⊃ <WFF>
                        | <WFF> , <WFF>    .
```

This proposal has the nasty effect that the induction rule of LCF has to restricted in scope. There would have to be some syntactic check made on wffs to determine whether they admit induction. Igarashi has studied this problem in [11].

11.5.1. Derived Deduction Rules

With a high level command language as we proposed and with the richer implicative structure that we are now discussing, one is able to write **in** the logic, rules of the form below which would have to be built into the system:

$$\frac{A \vdash B \qquad C \vdash D}{E \vdash F}$$

11.6. Concrete Syntax

The question of how LCF should deal with syntax of programming languages is rather important since we hope to apply the system to many languages. The problem is that we want to be able to specify the concrete syntax of a language so we can simply refer to a program by its text and have the system deduce its structure. We don't have a solid solution to the problem; in this thesis we have discussed some questions of denotation and syntax in relation to LISP and LAP but much more work is needed.

11.7. Extending Simplification

11.7.1. Inequalities

In chapter 8 we were unable to complete a proof because simplification, which is the workhorse of the system, only deals with equalities. It is a little more arkward to handle inequalities but the extent of the technical problems is the fact that applicability depends on which side of an AWFF is being simplified and whether the user is doing forward or backward reasoning. For example, $A \sqsubseteq B$ can be used to simplify a step $g(B) \sqsubseteq C$ to a step $g(A) \sqsubseteq C$ but not used to simplify a goal $F \sqsubseteq H(A)$. An important consideration is that one would like to simplify by equalities before inequalities which leads us to:

11.7.2. Split Level Simplification

There are many reasons why users want some simplification rules tried before others. The most notable is recursive function definitions which should usually be considered last-resort rules. One approach which at least deserves trial-by-experience is the idea of having two or more levels of simpset. The highest level will contain rules which have complicated conditions to check before they may be applied or which may lead to excessive expansion of the formula if applied several times without lower level rules intervening

11.7.3. n-time Simplification

Another facility which is an old idea (Weyhrauch) is that of having a counter on simplification rules which enable a user to specify that a certain rule (perhaps recursive) should only be used a limited number of times.

11.7.4. Subgoals from Conditional Simplification

When simplification is used as a tactic (i.e. to attack a goal), the user should be able to nominate certain conditional simplification rules which are always applied when the left hand side matches; conditions are still attacked by simplification but those that are not reduced to trivialities are made into subgoals. It is necessary to specifically nominate rules to have this property (globally or locally) to avoid generation of large numbers of false subgoals.

11.7.5. Case Analysis in Simplification

Suppose we are given terms A, $\mathbf{F}$ and $\mathbf{G}$ where $\mathbf{F}$ and $\mathbf{G}$ contain occurrences of A. We propose that simplification should normally mutate the term $A \rightarrow \mathbf{F},\mathbf{G}$ through $A \rightarrow \mathbf{F}\{\mathbb{T}/A\},\mathbf{G}\{\mathbb{F}/A\}$ to perhaps something simpler. (recall that T{s/x} denotes the result of substituting s for x in T.)

11.7.6. Simplifying Procedures

In the current system members of the simpset have the form $C \vdash A \equiv B$ and if A matches some subterm and C is satisfied (by recursive call on simplification) then B (appropriately modified) replaces the matched subterm. We propose a more general scheme where items in the simpset are triples (A,C,F). As before the term A must match a subterm before any consideration is given to the item; following a match, condition C is checked (by some procedure) and if it is OK then function F (given in some language) is executed with the matched subterm as a parameter.

11.8. Types

In his original suggestion and formulation of the pure logic in [1], Scott chose a typed version since he despaired of finding a model for the λ-calculus and concluded, on this basis, that 'the theory of types is here to stay'. Since that time, Scott has produced models for the λ-calculus in [8], has repudiated the 'OWHY paper’ and has formulated a type-free logic ([9]). This development inevitably raises the question as to whether a new LCF system should be typed or not.

Now, if we make the new LCF typed, we can apply some lessons learnt from the old system. Foremost, the system should be made to check types of terms. More precisely, the system should check that a proof is consistently typable; the user should rarely have to actually specify the types explicitly. The fact that the old system did not do this can be justified on the grounds that it was the prototype but this argument does not apply now. Next lesson is that the pure logic should be changed to allow an arbitrary number of base domains, instead of just D_{ind} and D_{tr} . In using the current system, where D_{ind} must be partitioned into various notional data types (such as integers and lists), one's theorems tend to be cluttered up with relativisations; Also, many theorems only exist because the data domains are only notional. Then, if we have many base types, we must also think about a richer type structure: namely, if α and β are types then $\alpha \to \beta$ (as before), $\alpha + \beta$ (disjoint union) and $\alpha * \beta$ (cartesian product) should be too.

If, on the other hand, the new LCF implements Scott's type-free logic, one must provide syntactic sugar with which the user may restrict terms to certain subdomains (by means of hidden retractions) to achieve notional data-types. It must be noted that the provision of this facility corresponds approximately, in difficulty, to building type-checking into a typed system.

The debate continues as to which of these options is best. It cannot be denied that the type-free logic is mathematically more elegant. Also, some people say that one can more easily axiomatise programming languages with functional data types using it. On the other hand, some people say that the objects we deal with in computation are really well-typed and that when one is proving properties of computable objects one should be forced to recognise the type structure.

11.9. Miscellaneous Improvements

11.9.1. Solving Equations

We found it convenient in doing the LISP experiments to have various theorems available with the flavor of

$$P \to \mathbb{F}, q \equiv \mathbb{T} \vdash P \equiv \mathbb{F},\ q \equiv \mathbb{T}$$

and

$$P \to \mathbb{T}, \perp \equiv \mathbb{T} \vdash P \equiv \mathbb{T}.$$

With a couple of dozen such theorems one can break down some quite complex equations to give specific truth-values for some of the subterms of the original equation. For example

$$p \to \mathbb{T}, (q \to \perp, (r \to \mathbb{T}, s)) \equiv \mathbb{F}$$

may be solved for p,q,r,s (in this case each is $\mathbb{F}$).

This process, which we call 'solving equations' is clearly one one which should be automated.

11.9.2. Definitional Facilities

In the current system, if one wants to name (with identifier n, say) some complicated term T that is used often in the proof of some step S but does not actually appear in the step, then one can either make the WFF $n \equiv T$ an axiom or an assumption. In one case one gets to complicate the axioms unecessarily and in the other case $n \equiv T$ becomes a dependency of S. This deficiency must be removed in the next system.

11.9.3. Automatic Forward Reasoning

We propose to have a set of sentences, called an FR-set, and a mechanism called 'Consequences'. When Consequences is invoked with a set of steps, antecedents of sentences in the FR-set are checked for satisfiability by the nominated steps. If all antecedents of a sentence check out, then the consequent (with appropriate instantiation) is made into a new step in the proof and perhaps added to the simpset. It should be clear that if a step so generated happens to be a standard contradiction then the current goal will be established.

11.9.4. More Abbreviations

The universal quantifier and relativisation (split arrow) abbreviations of LCF have been very successful. It seems that abbreviating the term $P \to \mathbb{F},\mathbb{T}$ as $\neg P$ would also be extremely useful.

We propose also that empirical study be devoted to having $P \wedge Q$ as an abbreviation for one of the terms $P \to Q,\mathbb{F}$ and $P \to Q,Q \to \mathbb{F},\mathbb{F}$. Similarly, $P \vee Q$ could be a abbreviation for one of the terms $P \to \mathbb{T},Q$ and $P \to (Q \to \mathbb{T},\mathbb{T}),Q$.

CHAPTER 12

Conclusion

This thesis has been an extensive exercise in the application of LCF to the definition of some programming languages - a subset of a machine-language and some subsets of LISP. In each of the several cases, we have **defined** the language axiomatically but have also illustrated how a 'theory' for the language should be constructed using the axioms as a base. The theory of a language then becomes a framework in which programs of the language can be proved correct.

[23] classifies methods of definition of semantics as being either 'constructive' and suited to the needs of the implementor or 'implicit' and suited to the needs of the user. It then argues that a language should be defined both ways and the definitions proved consistent. In the present work, the definition of Pure LISP, for example, is clearly constructive but many of the theorems of the Theory of Pure LISP have the flavour of rules in Hoare's method ([20]). It would be interesting to investigate whether some subset of theorems of the Theory of Pure LISP could be used as a satisfactory implicit definition.

We note that the recent independent work of M. Gordon [16] also gives a semantics of Pure LISP leading to a proof of correctness of 'eval' etc. We observe several significant differences of approach which make his work and this thesis somewhat complementary. Gordon ascribes denotations directly to Pure LISP M-expressions using Scott/Strachey style semantic equations (as in [29]) whereas we have it that S-expressions denote functions under interpretation of a particular 'eval' function written in LCF. Gordon's approach makes use of much more logical machinery than is available in LCF and so his proofs are not checkable mechanically (as yet). Machine checkability was a prime requirement in this thesis since automation is the ultimate goal of the project.

The way we were able to separate syntax from semantics by means of notation and denotation considerations is a technique that hopefully could be applied with benefit to other languages; certainly, it solved the problem completely in the cases we studied.

As an experiment in the application of LCF to the specification of programming language semantics, the work was very encouraging. The logic has distinguished itself as regards expressive power; the actual definitions of the various languages are concise and elegant. It is true that in the case of Pure LISP the language being defined and the formalism are similar in structure but LAP is certainly different in structure to LCF and current work on an axiomatisation of PASCAL by Aiello et al ([10]) is proceeding well.

It is worth noting that we were able, in the case of LAP, to give a partial specification of a language and also, as in the case of Pure LISP, give a complete description of a language.

Although LCF is yet in its infancy of development, it has already proved very suitable for discussion of Pure LISP programs. We would like to claim this is some evidence that LCF has a bright future in the area of program correctness. There are many aspects of the LCF system which have helped substantially in proof generation but the proofs cry out for more mechanisation and more powerful deduction rules.

We claim that this point in time is the end of the first cyle of development for LCF. Clearly the time is ripe for developing a brand new LCF system which incorporates the suggestions we have presented. Effort spent in this direction should generate the most payoff. After that is done, a revamping of the work on integers, lists and finite sets would be profitable since the axiomatisations could be polished somewhat. Also it would give a good measure of the improvement in deductive power between the two generations.

Redoing the Pure LISP proofs and completing the proof of correctness of the Pure LISP interpreter on the machine is a must and another look at the correctness of LCom0 would be appropriate. An option to be kept in mind at that time would be the reduction of the subset of LISP that LCom0 is written in and compiles. The AND, OR, and NOT features could be removed and that would simplify the compiler substantially but not to the point of no interest. However if the increase in power in new LCF lives up to hopes, this will not be necessary. In fact, we would expect to be able to attack the LCom4 compiler mentioned in [13], although our present treatment of LAP would then be inadequate.

The compiler proof has a number of disadvantages as an experiment using LCF. Most important of these is that it encourages work on a rather artificial subset of LISP and gross simplifications of PDP10 code. It would seem more fruitful to pick an experiment which would encourage, instead, more sophisticated theories of a low level language or a high level language.

APPENDIX 1

Theorems of LCom0 LISP

In this appendix we report on the Theory of LCom0 LISP that was developed as background for the compiler proof. The axioms on which this collection of theorems is based are given in the first 2 sections of Chapter 9. Note that practically all the results are suitable for direct inclusion in a SIMPSET.

The Interpreting Functions:

In this section we present theorems to do with the LCF functions of the interpretive semantics for LCom0 LISP - namely, 'eval', 'evcon', 'evand', 'evor', 'apply', 'evlis' and 'pairlis' (in that order):

$\vdash \forall x\ y.\ \mathrm{eval}(\bot,x,y) \equiv \bot$
$\vdash \forall x\ y.\ \mathrm{eval}(x,\bot,y) \equiv \bot$
$\vdash \forall x\ y.\ \mathrm{eval}(x,y,\bot) \equiv \bot$
$\partial(\mathrm{eval}(x,v,f)) \equiv \mathbb{T} \ \vdash\ \partial(x) \equiv \mathbb{T}$
$\partial(\mathrm{eval}(x,v,f)) \equiv \mathbb{T} \ \vdash\ \partial(v) \equiv \mathbb{T}$
$\partial(\mathrm{eval}(x,v,f)) \equiv \mathbb{T} \ \vdash\ \partial(f) \equiv \mathbb{T}$
$\vdash \forall v\ f.\ \mathrm{eval}(\mathrm{NIL},v,f) \equiv \partial(v)\rightarrow(\partial(f)\rightarrow\mathrm{NIL},\bot),\bot$
$\mathrm{isint}(x) \equiv \mathbb{T} \ \vdash\ \forall v\ f.\ \mathrm{eval}(x,v,f) \equiv \partial(v)\rightarrow(\partial(f)\rightarrow x,\bot),\bot$
$\mathrm{isname}(x) \equiv \mathbb{T},\ \mathrm{islist}(vb) \equiv \mathbb{T},\ \partial(FL) \equiv \mathbb{T}$
$\qquad\qquad \vdash\ \forall y.\ \mathrm{eval}(x,(x\cdot y)\cdot vb,fl) \equiv y$

$$\begin{aligned}&\text{isname}(x) \equiv \mathbb{T},\ \partial(y1) \equiv \mathbb{T},\ x{=}x1 \equiv \mathbb{F},\ \text{islist}(vb) \equiv \mathbb{T},\ \partial(fl) \equiv \mathbb{T}\\ &\qquad\qquad\qquad \vdash\ \forall y.\ \text{eval}(x,(x1\cdot y1)\cdot((x\cdot y)\cdot vb),fl) \equiv y\end{aligned}$$

$$\begin{aligned}&\text{isname}(x) \equiv \mathbb{T},\ \partial(y1) \equiv \mathbb{T},\ \partial(y2) \equiv \mathbb{T},\ x{=}x1 \equiv \mathbb{F},\\ &\qquad x{=}x2 \equiv \mathbb{F},\ \text{islist}(vb) \equiv \mathbb{T},\ \partial(fl) \equiv \mathbb{T}\\ &\qquad\qquad \vdash\ \forall y.\ \text{eval}(x,(x1\cdot y1)\cdot((x2\cdot y2)\cdot((x\cdot y)\cdot vb)),fl) \equiv y\end{aligned}$$

$$\partial(vb) \equiv \mathbb{T},\ \partial(fl) \equiv \mathbb{T}\ \vdash\ \forall x\ .\ \text{eval}(\text{QUOTE}\cdot(x\cdot \text{NIL}),vb,fl) \equiv x$$

$$\vdash\ \forall x\ vb\ fl.\ \text{eval}(\text{COND}\cdot x,vb,fl) \equiv \text{evcon}(x,vb,fl)$$

$$\vdash\ \forall x\ vb\ fl.\ \text{eval}(\text{AND}\cdot x,vb,fl) \equiv \partial(vb)\rightarrow(\partial(fl)\rightarrow\text{evand}(x,vb,fl),\bot),\bot$$

$$\vdash\ \forall x\ vb\ fl.\ \text{eval}(\text{OR}\cdot x,vb,fl) \equiv \partial(vb)\rightarrow(\partial(fl)\rightarrow\text{evor}(x,vb,fl),\bot),\bot$$

$$\vdash\ \forall x\ y.\ \text{evcon}(\bot,x,y) \equiv \bot$$

$$\vdash\ \forall x\ y.\ \text{evcon}(x,\bot,y) \equiv \bot$$

$$\vdash\ \forall x\ y.\ \text{evcon}(x,y,\bot) \equiv \bot$$

$$\vdash\ \forall vb\ fl.\ \text{evcon}(\text{NIL},vb,fl) \equiv \bot$$

$$\vdash\ \forall x\ y\ vb\ fl.\ \text{evcon}(((\text{QUOTE}\cdot(\text{T}\cdot\text{NIL}))\cdot(x\cdot\text{NIL}))\cdot y,vb,fl) \equiv \partial(y)\rightarrow\text{eval}(x,vb,fl),\bot$$

$$\begin{aligned}&\vdash\ \forall x\ y\ w\ vb\ fl.\ \text{evcon}((w\cdot(x\cdot\text{NIL}))\cdot y,vb,fl) \equiv \partial(x)\rightarrow(\partial(y)\rightarrow\\ &\qquad (\text{null}(\text{eval}(w,vb,fl))\rightarrow\text{evcon}(y,vb,fl),\text{eval}(x,vb,fl)),\bot),\bot\end{aligned}$$

$$\vdash\ \forall x\ y.\ \text{evand}(\bot,x,y) \equiv \bot$$

$$\vdash\ \forall vb\ fl.\ \text{evand}(\text{NIL},vb,fl) \equiv \text{T}$$

$$\begin{aligned}&\vdash\ \forall x\ y\ vb\ fl.\ \text{evand}(x\cdot y,vb,fl)\\ &\qquad\qquad \equiv \partial(y)\rightarrow(\text{null}(\text{eval}(x,vb,fl))\rightarrow\text{NIL},\text{evand}(y,vb,fl)),\bot\end{aligned}$$

$$\vdash\ \forall x\ y.\ \text{evor}(\bot,x,y) \equiv \bot$$

$$\vdash\ \forall vb\ fl.\ \text{evor}(\text{NIL},vb,fl) \equiv \text{NIL}$$

$$\begin{aligned}&\vdash\ \forall x\ y\ vb\ fl.\ \text{evor}(x\cdot y,vb,fl)\\ &\qquad\qquad \equiv \partial(y)\rightarrow(\text{null}(\text{eval}(x,vb,fl))\rightarrow\text{evor}(y,vb,fl),\text{T}),\bot\end{aligned}$$

$$\vdash\ \forall x\ vb\ fl.\ \text{apply}(\bot,x,vb,fl) \equiv \bot$$

$$\vdash\ \forall fn\ vb\ fl.\ \text{apply}(fn,\bot,vb,fl) \equiv \bot$$

$$\vdash\ \forall fn\ x\ fl.\ \text{apply}(fn,x,\bot,fl) \equiv \bot$$

$$\vdash\ \forall fn\ x\ vb.\ \text{apply}(fn,x,vb,\bot) \equiv \bot$$

$$\partial(\text{apply}(fn,x,vb,fl)) \equiv \mathbb{T}\ \vdash\ \partial(fn) \equiv \mathbb{T}$$

$$\partial(\text{apply}(fn,x,vb,fl)) \equiv \mathbb{T}\ \vdash\ \partial(x) \equiv \mathbb{T}$$

$$\partial(\text{apply}(fn,x,vb,fl)) \equiv \mathbb{T}\ \vdash\ \partial(vb) \equiv \mathbb{T}$$

$$\partial(\text{apply}(fn,x,vb,fl)) \equiv \mathbb{T}\ \vdash\ \partial(fl) \equiv \mathbb{T}$$

$$\vdash\ \forall vb\ fl.\ \mathrm{evlis}(\bot,vb,fl) \equiv \bot$$

$$\mathrm{islist}(x) \equiv \mathbb{F}\quad \vdash\ \forall vb\ fl.\ \mathrm{evlis}(x,vb,fl) \equiv \bot$$

$$\vdash\ \forall vb\ fl.\ \mathrm{evlis}(\mathrm{NIL},vb,fl) \equiv \mathrm{NIL}$$

$$\vdash\ \forall x\ vb\ fl.\ \mathrm{evlis}(x\circ\mathrm{NIL},vb,fl) \equiv \mathrm{eval}(x,vb,fl)\circ\mathrm{NIL}$$

$$\begin{aligned}\vdash\ \forall x1\ x2\ vb\ fl.\ &\mathrm{evlis}(x1\circ(x2\circ\mathrm{NIL}),vb,fl)\\ &\equiv \mathrm{eval}(x1,vb,fl)\circ(\mathrm{eval}(x2,vb,fl)\circ\mathrm{NIL})\end{aligned}$$

$$\begin{aligned}\vdash\ \forall x1\ x2\ x3\ vb\ fl.\ &\mathrm{evlis}(x1\circ(x2\circ(x3\circ\mathrm{NIL})),vb,fl)\\ &\equiv \mathrm{eval}(x1,vb,fl)\circ(\mathrm{eval}(x2,vb,fl)\circ(\mathrm{eval}(x3,vb,fl)\circ\mathrm{NIL}))\end{aligned}$$

$$\vdash\ \forall x\ a.\ \mathrm{pairlis}(\bot,x,a) \equiv \bot$$

$$\vdash\ \forall x\ y.\ \mathrm{pairlis}(x,y,\bot) \equiv \bot$$

$$\vdash\ \forall x\ a.\ \mathrm{pairlis}(\mathrm{NIL},x,a) \equiv a$$

$$\vdash\ \forall x\ y\ a.\ \mathrm{pairlis}(x\circ\mathrm{NIL},y\circ\mathrm{NIL},a) \equiv (x\circ y)\circ a$$

$$\begin{aligned}\vdash\ \forall x1\ x2\ y1\ y2\ a.\ &\mathrm{pairlis}(x1\circ(x2\circ\mathrm{NIL}),y1\circ(y2\circ\mathrm{NIL}),a)\\ &\equiv (x1\circ y1)\circ((x2\circ y2)\circ a)\end{aligned}$$

$$\begin{aligned}\vdash\ \forall x1\ x2\ x3\ y1\ y2\ y3\ a\ .\ &\mathrm{pairlis}(x1\circ(x2\circ(x3\circ\mathrm{NIL})),y1\circ(y2\circ(y3\circ\mathrm{NIL})),a)\\ &\equiv (x1\circ y1)\circ((x2\circ y2)\circ((x3\circ y3)\circ a))\end{aligned}$$

The Built-In Functions:

Presented here are the effects of applying 'eval' to expressions of the form $\mathbf{F}\circ\mathbf{X}$ (where $\mathbf{F}$ is a built-in function) and applying 'apply' to built in functions and suitable argument lists.

$$\vdash\ \forall x\ vb\ fl.\ \mathrm{apply}(\mathrm{CAR},x\circ\mathrm{NIL},vb,fl) \equiv \partial(vb)\rightarrow(\partial(fl)\rightarrow \mathrm{hd}(x),\bot),\bot$$

$$\vdash\ \forall x\ vb\ fl.\ \mathrm{apply}(\mathrm{CDR},x\circ\mathrm{NIL},vb,fl) \equiv \partial(vb)\rightarrow(\partial(fl)\rightarrow \mathrm{tl}(x),\bot),\bot$$

$$\vdash\ \forall x\ vb\ fl.\ \mathrm{apply}(\mathrm{NOT},x\circ\mathrm{NIL},vb,fl) \equiv \partial(vb)\rightarrow(\partial(fl)\rightarrow(\mathrm{null}(x)\rightarrow \mathrm{T},\mathrm{NIL}),\bot),\bot$$

$$\vdash\ \forall x\ vb\ fl.\ apply(ATOM, x \cdot NIL, vb, fl) \equiv \partial(vb) \to (\partial(fl) \to (atom(x) \to T, NIL), \bot), \bot$$
$$\vdash\ \forall x\ y\ vb\ fl.\ apply(CONS, x \cdot (y \cdot NIL), vb, fl) \equiv \partial(vb) \to (\partial(fl) \to (x \cdot y), \bot), \bot$$
$$\vdash\ \forall x\ vb\ fl.\ apply(LIST, x, vb, fl) \equiv \partial(vb) \to (\partial(fl) \to x, \bot), \bot$$
$$\vdash\ \forall x\ y\ vb\ fl.\ apply(EQUAL, x \cdot (y \cdot NIL), vb, fl) \equiv \partial(vb) \to (\partial(fl) \to ((x = y) \to T, NIL), \bot), \bot$$
$$\vdash\ \forall x\ y\ vb\ fl.\ apply(PLUS, x \cdot (y \cdot NIL), vb, fl) \equiv \partial(vb) \to (\partial(fl) \to (x + y), \bot), \bot$$
$$\vdash\ \forall x\ y\ vb\ fl.\ apply(TIMES, x \cdot (y \cdot NIL), vb, fl) \equiv \partial(vb) \to (\partial(fl) \to (x * y), \bot), \bot$$
$$\vdash\ \forall x\ vb\ fl.\ apply(MINUS, x \cdot NIL, vb, fl) \equiv \partial(vb) \to (\partial(fl) \to mns(x), \bot), \bot$$
$$\vdash\ \forall x\ vb\ fl.\ apply(GENSYM, x \cdot NIL, vb, fl) \equiv \partial(vb) \to (\partial(fl) \to gensym(x), \bot), \bot$$
$$\vdash\ \forall x\ vb\ fl.\ apply(NUMBERP, x \cdot NIL, vb, fl) \equiv \partial(vb) \to (\partial(fl) \to (isint(x) \to T, NIL), \bot), \bot$$
$$\begin{aligned}\vdash\ \forall x\ y\ vb\ fl.\ & apply(GREATERP, x \cdot (y \cdot NIL), vb, fl) \\ & \equiv \partial(vb) \to (\partial(fl) \to ((x > y) \to T, NIL), \bot), \bot\end{aligned}$$

$$\vdash\ \forall x\ vb\ fl.\ eval(CAR \cdot (x \cdot NIL), vb, fl) \equiv hd(eval(x, vb, fl))$$
$$\vdash\ \forall x\ vb\ fl.\ eval(CDR \cdot (x \cdot NIL), vb, fl) \equiv tl(eval(x, vb, fl))$$
$$\vdash\ \forall x\ vb\ fl.\ eval(NOT \cdot (x \cdot NIL), vb, fl) \equiv null(x) \to T, NIL$$
$$\vdash\ \forall x\ vb\ fl.\ eval(ATOM \cdot (x \cdot NIL), vb, fl) \equiv atom(eval(x, vb, fl)) \to T, NIL$$
$$\vdash\ \forall x\ y\ vb\ fl.\ eval(CONS \cdot (x \cdot (y \cdot NIL)), vb, fl) \equiv eval(x, vb, fl) \cdot eval(y, vb, fl)$$
$$\vdash\ \forall x\ vb\ fl.\ eval(LIST \cdot x, vb, fl) \equiv evlis(x, vb, fl)$$
$$\begin{aligned}\vdash\ \forall x\ y\ vb\ fl.\ & eval(EQUAL \cdot (x \cdot (y \cdot NIL)), vb, fl) \\ & \equiv (eval(x, vb, fl) = eval(y, vb, fl)) \to T, NIL\end{aligned}$$
$$\begin{aligned}\vdash\ \forall x\ y\ vb\ fl.\ & eval(PLUS \cdot (x \cdot (y \cdot NIL)), vb, fl) \\ & \equiv eval(x, vb, fl) + eval(y, vb, fl)\end{aligned}$$
$$\begin{aligned}\vdash\ \forall x\ y\ vb\ fl.\ & eval(TIMES \cdot (x \cdot (y \cdot NIL)), vb, fl) \\ & \equiv eval(x, vb, fl) * eval(y, vb, fl)\end{aligned}$$
$$\vdash\ \forall x\ vb\ fl.\ eval(MINUS \cdot (x \cdot NIL), vb, fl) \equiv mns(eval(x, vb, fl))$$
$$\vdash\ \forall x\ vb\ fl.\ eval(GENSYM \cdot (x \cdot NIL), vb, fl) \equiv gensym(eval(x, vb, fl))$$
$$\vdash\ \forall x\ vb\ fl.\ eval(NUMBERP \cdot (x \cdot NIL), vb, fl) \equiv isint(eval(x, vb, fl)) \to T, NIL$$
$$\begin{aligned}\vdash\ \forall x\ y\ vb\ fl.\ & eval(GREATERP \cdot (x \cdot (y \cdot NIL)), vb, fl) \\ & \equiv (eval(x, vb, fl) > eval(y, vb, fl)) \to T, NIL\end{aligned}$$

LAMBDA Expressions:

Here we give the effect of 'eval'ing and 'apply'ing LAMBDA expressions.

$$\vdash\ \forall b\ vb\ fl.\ apply(\{LAMBDA\ NIL\ b\}, NIL, vb, fl) \equiv eval(b, vb, fl)$$
$$\begin{aligned}\vdash\ \forall x\ y\ b\ vb\ fl.\ & apply(\{LAMBDA\ \{x\}\ b\}, \{y\}, vb, fl) \\ & \equiv eval(b, (x \cdot y) \cdot vb, fl)\end{aligned}$$

$$\begin{aligned}&\vdash \forall x1\ x2\ y1\ y2\ b\ vb\ fl.\ \mathrm{apply}(\{\mathrm{LAMBDA}\ \{x1\ x2\}\ b\},\{y1\ y2\},vb,fl)\\&\qquad\equiv \mathrm{eval}(b,(x1\circ y1)\circ((x2\circ y2)\circ vb),fl)\end{aligned}$$

$$\begin{aligned}&\vdash \forall x1\ x2\ x3\ y1\ y2\ y3\ b\ vb\ fl.\ \mathrm{apply}(\{\mathrm{LAMBDA}\ \{x1\ x2\ x3\}\ b\}\\&\qquad\qquad,\{y1\ y2\ y3\},vb,fl)\\&\qquad\equiv \mathrm{eval}(b,(x1\circ y1)\circ((x2\circ y2)\circ((x3\circ y3)\circ vb)),fl)\end{aligned}$$

$$\vdash \forall b\ vb\ fl.\ \mathrm{eval}(\{\mathrm{LAMBDA\ NIL}\ b\}\circ \mathrm{NIL},vb,fl) \equiv \mathrm{eval}(b,vb,fl)$$

$$\begin{aligned}&\vdash \forall x\ y\ b\ vb\ fl.\ \mathrm{eval}(\{\mathrm{LAMBDA}\ \{x\}\ b\}\circ\{y\},vb,fl)\\&\qquad\equiv \mathrm{eval}(b,\ (x\circ \mathrm{eval}(y,vb,fl))\circ vb,\ fl)\end{aligned}$$

$$\begin{aligned}&\vdash \forall x1\ x2\ y1\ y2\ b\ vb\ fl.\ \mathrm{eval}(\{\mathrm{LAMBDA}\ \{x1\ x2\}\ b\}\circ\{y1\ y2\},vb,fl)\\&\qquad\equiv \mathrm{eval}(b,\ (x1\circ \mathrm{eval}(y1,vb,fl))\circ((x2\circ \mathrm{eval}(y2,vb,fl))\circ vb),\ fl)\end{aligned}$$

$$\begin{aligned}&\vdash \forall x1\ x2\ x3\ y1\ y2\ y3\ b\ vb\ fl.\\&\qquad\qquad \mathrm{eval}(\{\mathrm{LAMBDA}\ \{x1\ x2\ x3\}\ b\}\circ\{y1\ y2\ y3\},vb,fl)\\&\qquad\equiv \mathrm{eval}(b,\ (x1\circ \mathrm{eval}(y1,vb,fl))\circ((x2\circ \mathrm{eval}(y2,vb,fl))\\&\qquad\qquad\circ((x3\circ \mathrm{eval}(y3,vb,fl))\circ vb)),\ fl)\end{aligned}$$

The Basic Functions:

Here we give the meanings (under interpretation) of the basic LISP functions defined in Fig. 5.4:

$$\begin{aligned}\mathrm{BFD}(FL) \equiv \mathbb{T}\ \vdash\ &\mathrm{assoc}(\mathrm{NULL},FL) \equiv \mathrm{Snull},\\&\mathrm{assoc}(\mathrm{DIFFERENCE},FL) \equiv \mathrm{SdiFFerence},\\&\mathrm{assoc}(\mathrm{ISLIST},FL) \equiv \mathrm{Sislist},\\&\mathrm{assoc}(\mathrm{ASSOC},FL) \equiv \mathrm{Sassoc},\\&\mathrm{assoc}(\mathrm{LENGTH},FL) \equiv \mathrm{Slength},\\&\mathrm{assoc}(\mathrm{APPEND},FL) \equiv \mathrm{Sappend}\end{aligned}$$

$$\mathrm{BFD}(FL) \equiv \mathbb{T}\ \vdash\ \mathrm{islist}(FL) \equiv \mathbb{T}$$

$$\begin{aligned}\mathrm{BFD}(FL) \equiv \mathbb{T}\ \vdash\ &\forall e\ vb.\ \mathrm{apply}(\mathrm{NULL},e\circ \mathrm{NIL},vb,FL)\\&\qquad\equiv \mathrm{islist}(vb)\rightarrow(\mathrm{null}(e)\rightarrow T,\mathrm{NIL}),\perp\end{aligned}$$

$$\begin{aligned}\text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall e\ \text{vb. eval(NULL}\circ(e\circ\text{NIL}),\text{vb},\text{FL})\\ & \quad \equiv \text{null(eval(}e,\text{vb},\text{FL}))\to\text{T},\text{NIL}\end{aligned}$$

$$\begin{aligned}\text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall x\ y\ \text{vb. apply(DIFFERENCE},x\circ(y\circ\text{NIL}),\text{vb},\text{FL})\\ & \quad \equiv \text{islist(vb)}\to(x-y),\bot\\ \text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall x\ y\ \text{vb. eval(DIFFERENCE}\circ(x\circ(y\circ\text{NIL})),\text{vb},\text{FL})\\ & \quad \equiv \text{eval}(x,\text{vb},\text{FL})-\text{eval}(y,\text{vb},\text{FL})\end{aligned}$$

$$\begin{aligned}\text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall e\ \text{vb. apply(ISLIST},e\circ\text{NIL},\text{vb},\text{FL})\\ & \quad \equiv \text{islist(vb)}\to(\text{islist}(e)\to\text{T},\text{NIL}),\bot\\ \text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall e\ \text{vb. eval(ISLIST}\circ(e\circ\text{NIL}),\text{vb},\text{FL})\\ & \quad \equiv \text{islist(eval(}e,\text{vb},\text{FL}))\to\text{T},\text{NIL}\end{aligned}$$

$$\begin{aligned}\text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall x\ y\ \text{vb. apply(ASSOC},x\circ(y\circ\text{NIL}),\text{vb},\text{FL})\\ & \quad \equiv \text{islist(vb)}\to\text{assoc}(x,y),\bot\\ \text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall x\ y\ \text{vb. eval(ASSOC}\circ(x\circ(y\circ\text{NIL})),\text{vb},\text{FL})\\ & \quad \equiv \text{assoc(eval(}x,\text{vb},\text{FL}),\text{eval}(y,\text{vb},\text{FL}))\end{aligned}$$

$$\begin{aligned}\text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall e\ \text{vb. apply(LENGTH},e\circ\text{NIL},\text{vb},\text{FL})\\ & \quad \equiv \text{islist(vb)}\to\text{length}(e),\bot\\ \text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall e\ \text{vb. eval(LENGTH}\circ(e\circ\text{NIL}),\text{vb},\text{FL})\\ & \quad \equiv \text{length(eval(}e,\text{vb},\text{FL}))\end{aligned}$$

$$\begin{aligned}\text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall x\ y\ \text{vb. apply(APPEND},x\circ(y\circ\text{NIL}),\text{vb},\text{FL})\\ & \quad \equiv \text{islist(vb)}\to(x\,\&\,y),\bot\\ \text{BFD(FL)} \equiv \mathbb{T} \vdash{} & \forall x\ y\ \text{vb. eval(APPEND}\circ(x\circ(y\circ\text{NIL})),\text{vb},\text{FL})\\ & \quad \equiv \text{eval}(x,\text{vb},\text{FL})\,\&\,\text{eval}(y,\text{vb},\text{FL})\end{aligned}$$

APPENDIX 2

Yet Another LISP Subset

In Figure A2.1 (next three pages) we give an interpretive semantics for yet another subset of LISP - a superset of Pure LISP which has **SET**s, **SETQ**s, **PROG**s, **GENSYM**s and property lists as well as the **AND**, **OR**, **NOT** and **LIST** operations introduced in LCom0 LISP. This semantics includes all the techniques that we discussed while developing the other versions of LISP.

The 'eval' and 'apply' functions in the definition of Pure LISP had a parameter which was an A-list for holding the values of bound variables. The corresponding functions in this treatment have a 'state' parameter instead; A state is a triple of A-list (for bound variable values), list of property lists of variables and memory for the gensym function. To allow for side-effects, each of the functions ('eval', 'apply', etc.) returns as a pair, the regular answer and a new state.

```
**AXIOM NL1:

      lisp ≡ [λe. hd(eval(e,NIL·(NIL·G0001)))]

      eval ≡ [μB. evalF(B)],
      evalF ≡ [λB x st. ∂(st) →
                    null(x) → NIL·st,
                    isint(x) → x·st,
                    isname(x)→[λy. null(y)→tl(assoc(VALUE,tl(assoc(x,tl(hd(st))))))·st,
                                             tl(y)·st] (assoc(x,hd(hd(st)))),
                    atom(x) → ⊥,
                    hd(x)=QUOTE → hd(tl(x))·st,

                    hd(x)=COND  → [μG.evconF(B,G)](tl(x),st),
                    hd(x)=AND   → [μG.evandF(B,G)] (tl(x),st),
                    hd(x)=OR    → [μG.evorF(B,G)] (tl(x),st),
                    hd(x)=PROG  → [μG.evprog(B,G)]
                                   (tl(tl(x)), initvars(hd(tl(x)),hd(st)), tl(st)),

                    hd(x)=GENSYM→ [λz. z • (hd(st)·(hd(tl(st))·z))]
                                           (gensym(tl(tl(st)))),

                    hd(x)=SETQ  → [λvst.
                                   [λvar val st1.
                                      [λal pl gm.
                                         [λy. null(y) →
                                             val·(al·(put(val,var,VALUE,pl)·gm)),
                                             val·(set(var,val,al)·(pl·gm))]
                                         (assoc(var,al))]
                                      (hd(st1),hd(tl(st1)),tl(tl(st1)))]
                                   (hd(tl(x)),hd(vst),tl(vst))]
                                (B(hd(tl(tl(x))),st)),

                    [λz. [μG.applyF(B,G)](hd(x),hd(z),tl(z))]
                              ([μG.evlisF(B,G)](tl(x),st)),⊥],
```

Figure A2.1a - Axioms for Yet Another LISP.

```
evcon ≡ [μG. evconF(eval,G)],
evconF ≡ [λF G x st. [λz. null(hd(z))→F(tl(x),tl(z)),
                G(hd(tl((hd(x))),tl(z))](F(hd(hd(x)),st))],

evand ≡ [μG. evandF(eval,G)],
evandF ≡ [λF G x st. null(x)→T, [λz. null(hd(z))→NIL,G(tl(x),tl(z))]
                                                    (F(hd(x),st))],
evor  ≡ [μG. evorF(eval,G)],
evorF ≡ [λF G x st. null(x)→NIL, [λz. null(hd(z))→G(tl(x),tl(z)),T]
                                                    (F(hd(x),st))],
evlis ≡ [μG. evlisF(eval,G)],
evlisF ≡ [λF G m vb fl. null(m)→NIL∘st,
               [λx.[λy. (hd(x)∘hd(y)) ∘ tl(y)] (G(tl(m),tl(x))]
                                                    (F(hd(m),st))]],
evprog ≡ [μG. evprogF(eval,G)],
evprogF ≡ [λF G m vb fl. null(m)→NIL∘st, [λx. G(tl(m),tl(x)](F(hd(m),st))],

apply ≡ [μG. applyF(eval,G)],
applyF ≡ [λF G fn x st. ∂(x) → ∂(st) →
              (fn=LIST) → x∘st,
              (fn=SET) → hd(tl(x))∘set(hd(x),hd(tl(x)),st),
              (fn=GET) → get(hd(x),hd(tl(x)),hd(tl(st))∘st,
              (fn=PUT) → hd(x)∘(hd(st)∘
                          (put(hd(x),hd(tl(x)),hd(tl(tl(x))),hd(tl(st)))∘tl(tl(st)))),
              isBF(fn) → applyBF(fn,x)∘st,
              isname(fn) → G(hd(F(x,st)),x,st),
              (hd(fn)=LAMBDA)→ [λz. hd(z) ∘
                             (prune(hd(tl(z)),hd(st))∘tl(tl(z)))]
                 (F(hd(tl(tl(fn))),pairlis(hd(tl(fn)),x,hd(st))∘tl(st))),
              (hd(fn)=LABEL)→ [λz. hd(z)∘ (tl(hd(tl(z))∘tl(tl(z)))]
                    (G(hd(tl(tl(fn))), x,
                       ((hd(tl(fn))∘hd(tl(tl(fn))))∘hd(st))∘tl(st))),
                                                 ⊥, ⊥, ⊥ ],
```

Figure A2.1b - Axioms for Yet Another LISP (ctd).

```
pairlis ≡ [μG.[λx y st. null(x) → (null(y) → st, ⊥),
                 [λz. ((hd(x)·hd(y))·hd(z)) • tl(z)](G(tl(x),tl(y),st))]]

prune ≡ [μG.[λx y. length(x)=length(y) → x, G(tl(x),y)],

set ≡ [μG.[λx y a. (hd(hd(a))=x) → (x·y)·tl(a), hd(a)·G(x,y,tl(a))]],

put ≡ [μG. [λvar val pn pl.
            null(pl) → (var·((pn·val)·NIL))·pl,
            (hd(hd(pl))=var) → (var·set(pn,val,tl(hd(pl))))·tl(pl),
                               hd(pl)·G(var,val,pn,tl(pl))]],

initvars ≡ [μG.[λvl al.null(vl) → al,(hd(vl)·NIL)·G(tl(vl),al)]],

get ≡ [μG. [λvar pn pl. null(pl) → NIL,
              (hd(hd(pl))=var) → [λz. null(z) → NIL,tl(z)]
                                      (assoc(pn,tl(hd(pl)))),
                                  G(var,pn,tl(pl))]],

isBF ≡  [λx. (x=CAR)→𝕋,    (x=CONS)→𝕋,    (x=MINUS)→𝕋,
             (x=CDR)→𝕋,    (x=PLUS)→𝕋,    (x=GENSYM)→𝕋,
             (x=NOT)→𝕋,    (x=EQUAL)→𝕋,   (x=NUMBERP)→𝕋,
             (x=ATOM)→𝕋,   (x=TIMES)→𝕋,   (x=GREATERP)]
applyBF(CAR) ≡ [λx. hd(hd(x))],
applyBF(CDR) ≡ [λx. tl(hd(x))],
applyBF(NOT) ≡ [λx. null(hd(x))→T,NIL],
applyBF(ATOM) ≡ [λx. atom(hd(x))→T,NIL],
applyBF(CONS) ≡ [λx. hd(x)·hd(tl(x))],
applyBF(PLUS) ≡ [λx. hd(x)+hd(tl(x))],
applyBF(EQUAL) ≡ [λx. hd(x)=hd(tl(x))→T,NIL],
applyBF(TIMES) ≡ [λx. hd(x)*hd(tl(x))],
applyBF(MINUS) ≡ [λx. mns(hd(x))],
applyBF(GENSYM) ≡ [λx. gensym(hd(x))],
applyBF(NUMBERP) ≡ [λx. isint(hd(x))→T,NIL],
applyBF(GREATERP) ≡ [λx. (hd(x)>hd(tl(x)))→T,NIL]
```

Figure A2.1c - Axioms for Yet Another LISP (ctd).

REFERENCES

LCF

[1] Scott, D. "**A Type-theoretical Alternative to CUCH, ISWIM, OWHY**", - (unpublished - now uncirculated) Oxford (1969).

[2] Milner, R. "**Implementation and Applications of Scott's Logic for Computable Functions**", Proc. A.C.M. Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, New Mexico, Jan. 1972.

[3] Milner, R. "**Logic for Computable Functions - Description of a Machine Implementation**", Artificial Intelligence Memo 169, Computer Science Dept., Stanford University, May 1972.

[4] Milner, R. "**Models of LCF**", Artificial Intelligence Memo 186, Computer Science Dept., Stanford University, Jan. 1973.

[5] Weyhrauch, R. & Milner, R. "**Program Semantics and Correctness in a mechanised Logic**", Proc. USA-Japan Computer Conference, Tokyo, Oct. 1972.

[6] Milner, R. & Weyhrauch, R. "**Proving Compiler Correctness in a Mechanised Logic**", Machine Intelligence 7, ed. D. Michie, Edinburgh University Press, 1972.

[7] Newey, M. "**Axioms and Theorems for Integers, Lists and Finite Sets in LCF**", Artificial Intelligence Memo 184, Computer Science Dept., Stanford University, March 1973.

[8] Scott, D., "**Lattice Theoretic Models for Various Type-Free Calculi**", Proc. 4th International Congress in Logic, Methodology and the Philosophy of Science, Bucharest, 1972.

[9] Scott, D., "**Data Types as Lattices**", Lecture Notes, Amsterdam, June 1972.

[10] Aiello, L., Aiello, M. & Weyhrauch, R.W., "**The Semantics of PASCAL in LCF**", Forthcoming A.I.Memo, Computer Science Dept., Stanford University.

[11] Igarashi, S., "**The Admissability of Fixed-Point Induction in First Order Logic of Typed Theories**", Artificial Intelligence Memo 168, Computer Science Dept., Stanford University, May 1972.

LISP

[12] McCarthy, J., Abrahams, P., Edwards, D., Hart, T. & Levin, M. **"LISP 1.5 Programmer's Manual"**, M.I.T. Press, 1962.

[13] London, R. **"Correctness of a Compiler for a LISP Subset"**, Proc. A.C.M. Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, New Mexico, Jan 1972.

[14] London, R. **"Correctness of Two Compilers for a LISP Subset"**, Artificial Intelligence Memo 151, Computer Science Dept., Stanford University, Oct. 1971.

[15] McCarthy, J. **"Recursive Functions of Symbolic Expressions and Their Computation by Machine"**, Comm. of A.C.M., Vol. 3, No. 4, (Apr 1960), pp 184-195.

[16] Gordon, M.J.C., **"Evaluation and Denotation of Pure LISP Programs; a Worked Example in Semantics"**, Ph.D. Thesis, School of Artificial Intelligence, Edinburgh University, 1974.

[17] Gordon, M.J.C., **"An Extended Abstract of "Models of Pure LISP" "**, Research Memo SAI-RM-7, School of Artificial Intelligence, Edinburgh University, Dec. 1973.

Semantics of Programming Languages

[18] de Bakker, J.W., **"Semantics of Programming Languages"**, Advances in Information Systems Science, Vol. 2, pp 173-227.

[19] Burstall, R.M., **"Formal Description of Program Structure and Semantics in First Order Logic"**, Machine Intelligence 5, Edinburgh University Press (1970), pp 79-98.

[20] Hoare, C.A.R., **"An Axiomatic Approach to Computer Programming"**, Comm. of A.C.M., Vol. 12, No. 10 (Oct 1969), pp 576-580, 583.

[21] Hoare, C.A.R., **"Procedures and Parameters: an Axiomatic Approach"**, Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, Vol. 188, Springer-Verlag, Berlin, pp 102-116.

[22] Hoare, C.A.R., **"Parallel Programming: an Axiomatic Approach"**, Artificial Intelligence Memo 219, Computer Science Dept., Stanford University, October 1973.

[23] Hoare, C.A.R. & Lauer, P.E., **"Consistent and Complementary Formal Theories of the Semantics of Programming Languages"**, Technical Report 44, Computing Laboratory, University of Newcastle upon Tyne, April 1973.

[24] Manna, Z., "**The Correctness of Programs**, J. Computer and System Sciences, 3 (1969), pp 119-127.

[25] McCarthy, J., "**Towards a Mathematical Science of Computation**", Proc. IFIP Congress, pp21-28, Amsterdam, North Holland (1962).

[26] McCarthy, J., "**A Formal Description of a Subset of Algol**", Proc. IFIP Working Conf. on "Formal Language Description Languages", North Holland, Amsterdam (1966).

[27] Mosses, P., "**The Mathematical Semantics of ALGOL 60**", Technical Monograph PRG-12, Oxford University Programming Research Group, Oxford (1974).

[28] Reynolds, J.C., "**On the Relation between Direct and Continuation Semantics**", Second Colloquium on Automata, Languages and Programming, Saarbrucken, (July 1974).

[29] Scott, D., and Strachey, C., "**Towards a Mathematical Semantics for Computer Languages**", Proceedings of the Symposium on Compters and automata, Microwave Research Institute Symposium Series, Vol 21, Vol 21.

[30] Waldinger, R. & Levitt, K.N., "**Reasoning about Programs**", Proc. ACM Sigact/Sigplan Symposium on Principles of Programming Language Design, Boston (1973).